

AD-A164 860

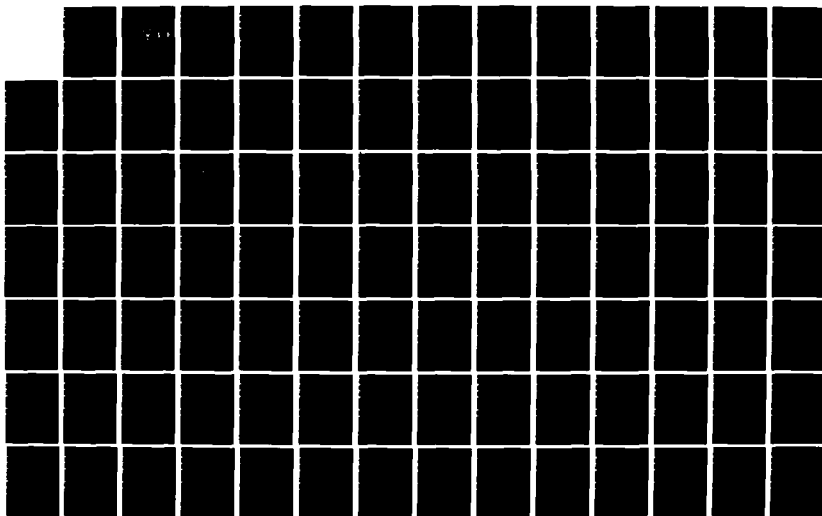
PROCESS SYNCHRONIZATION AND DATA COMMUNICATION BETWEEN
PROCESSES IN REAL TIME LOCAL AREA NETWORKS(U) NAVAL
POSTGRADUATE SCHOOL MONTEREY CA R HAEGER DEC 85

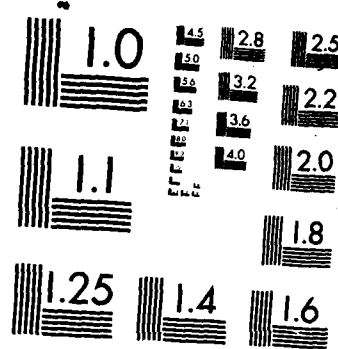
1/2

UNCLASSIFIED

F/G 17/2

NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

2

NAVAL POSTGRADUATE SCHOOL

Monterey, California

AD-A164 860



DTIC
ELECTE
MAR 05 1986
S D

THESIS

PROCESS SYNCHRONIZATION AND DATA
COMMUNICATION BETWEEN PROCESSES
IN REAL TIME LOCAL AREA NETWORKS

by

Reinhard Haeger

December 1985

Thesis Advisor:

Uno R. Kodres

Approved for public release; distribution is unlimited

DTIC FILE COPY

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED			1b. RESTRICTIVE MARKINGS		
2a. SECURITY CLASSIFICATION AUTHORITY			3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited		
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE			5. MONITORING ORGANIZATION REPORT NUMBER(S)		
4. PERFORMING ORGANIZATION REPORT NUMBER(S)			7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School		
6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School		6b. OFFICE SYMBOL (If applicable) 52		7b. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5100	
6c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5100		8a. NAME OF FUNDING/SPONSORING ORGANIZATION		9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
8b. OFFICE SYMBOL (If applicable)		8c. ADDRESS (City, State, and ZIP Code)		10. SOURCE OF FUNDING NUMBERS	
				PROGRAM ELEMENT NO.	PROJECT NO.
				TASK NO.	WORK UNIT ACCESSION NO.
11. TITLE (Include Security Classification) PROCESS SYNCHRONIZATION AND DATA COMMUNICATION BETWEEN PROCESSES IN REAL TIME LOCAL AREA NETWORKS					
12. PERSONAL AUTHOR(S) Haeger, Reinhard					
13a. TYPE OF REPORT Master's Thesis		13b. TIME COVERED FROM _____ TO _____		14. DATE OF REPORT (Year, Month, Day) 1985 December	
15. PAGE COUNT 113					
16. SUPPLEMENTARY NOTATION					
17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)		
FIELD	GROUP	SUB-GROUP	Distributed Processing, MULTIBUS, CP/M-86, Ethernet, Local Area Network, Shared Memory, Process Synchronization, Data Communication.		
19. ABSTRACT (Continue on reverse if necessary and identify by block number) This thesis extends the multi-computer real-time executive, MCORTEX. The multiple cluster system RTC* (Real Time Cluster Star), consisting of clusters of single board computers (INTEL iSBC 86/12A), which are connected via an Ethernet Local Area Network, serves as a hardware basis for the implementation of extended MCORTEX. The extension upgrades MCORTEX to system-wide synchronization and general data communication between any processes in the system. An intercluster shared memory model is developed, that partially replicates intracluster shared memory, such that shared data replication is minimized and the system's processing speed is maximized. This implementation, by transmitting produced shared data to all consuming clusters as soon as possible after production, guarantees that only cluster local hits occur in the system. Shared memory space is used (Cont)					
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS			21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED		
22a. NAME OF RESPONSIBLE INDIVIDUAL Prof. Uno R. Kodres			22b. TELEPHONE (Include Area Code) (408) 646-2197		22c. OFFICE SYMBOL 52Kr

19. ABSTRACT (Continued)

efficiently by transmitting shared data to consuming clusters only, and by the ability to store shared data contiguously in intracluster shared memory.

Approved for public release; distribution is unlimited.

Process Synchronization and Data Communication
between
Processes in Real Time Local Area Networks

by

Reinhard Haeger
Lieutenant Commander, Federal German Navy

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

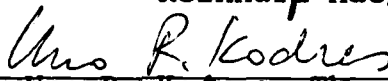
from the

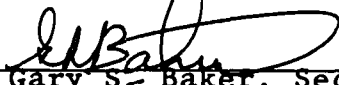
NAVAL POSTGRADUATE SCHOOL
December 1985


Author:



Reinhard Haeger

Approved by:


Uno R. Kodres, Thesis Advisor


Gary S. Baker, Second Reader


Vincent Y. Lum, Chairman,
Department of Computer Science


Kneale T. Marshall,
Dean of Information and Policy Sciences

ABSTRACT

This thesis extends the multi-computer real-time executive, MCORTEX. The multiple cluster system RTC* (Real Time Cluster Star), consisting of clusters of single board computers (INTEL iSBC 86/12A), which are connected via an Ethernet Local Area Network, serves as a hardware basis for the implementation of extended MCORTEX.

The extension upgrades MCORTEX to system-wide synchronization and general data communication between any processes in the system. An intercluster shared memory model is developed, that partially replicates intracluster shared memory, such that shared data replication is minimized and the system's processing speed is maximized.

This implementation, by transmitting produced shared data to all consuming clusters as soon as possible after production, guarantees that only cluster local hits occur in the system. Shared memory space is used efficiently by transmitting shared data to consuming clusters only, and by the ability to store shared data contiguously in intracluster shared memory.

DISCLAIMER

Some terms used in this thesis are registered trademarks of commercial products. Rather than attempt to cite each occurrence of a trademark, all trademarks appearing in this thesis will be listed below, following the firm holding the trademark:

1. INTEL Corporation, Santa Clara, California
INTEL
MULTIBUS
iSBC 86/12A
8086
2. Digital Research, Pacific Grove, California
PL/I-86
LINK86
ASM86
DDT86
3. XEROX Corporation, Stamford, Connecticut
Ethernet Local Area Network
4. InterLAN Corporation, Westford, Massachusetts
NI3010 Ethernet Communication Controller Board

TABLE OF CONTENTS

I.	INTRODUCTION	11
A.	DISCUSSION	11
	1. General	11
	2. Systems Architecture	11
	3. Specific	15
B.	BACKGROUND	15
C.	STRUCTURE OF THE THESIS	16
II.	ORGANIZATION OF INTERCLUSTER SHARED MEMORY	18
A.	SHARED MEMORY MODELS	18
	1. No Replication	18
	2. Total Replication	20
	3. Partial Replication	21
III.	ORGANIZATION OF INTRACLUSTER SHARED MEMORY	22
A.	CLASSES OF SHARED DATA	22
B.	SYSTEM SHARED DATA	23
C.	USER SHARED DATA	23
	1. Organization of Shared Data	24
	2. Storage of User Shared Data	25
D.	RELATION BETWEEN SHARED DATA	25
	1. The Ethernet Request Block	26
	2. User Shared Data Block	28
E.	INTRACLUSTER DATA FLOW	30
IV.	INTERCLUSTER DATA SHARING IN RTC*	33
A.	INTERCLUSTER CONNECTION	33
	1. The Ethernet	33
	2. The Ethernet Packet	34
B.	DRIVER - ECCB MESSAGE HANDOVER	37

1. Transmit Data Block	37
2. Receive Data Block	37
C. MESSAGE TRANSMISSION AND RECEPTION	39
1. The Relation Table	39
2. Data Format	42
3. Message Transmission	43
4. Message Reception	46
D. DATA SHARING	48
V. CONCLUSION	52
APPENDIX A: PROCEDURE MAKE_TABLE	54
APPENDIX B: PROCEDURE MAKE_MESSAGE	56
APPENDIX C: PROCEDURE PROCESS_PACKET	59
APPENDIX D: THE DRIVER	62
APPENDIX E: THE DEMONSTRATION PROGRAM	94
APPENDIX F: SYSTEM INITIALIZATION	108
LIST OF REFERENCES	111
INITIAL DISTRIBUTION LIST	112

13

Accession For	
NTIS CRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail and/or Special
A-1	

LIST OF TABLES

I	FILE RELATION.DAT	42
II	FILE SHARE.DCL	49
III	FILE POINTER.ASS	50

LIST OF FIGURES

1.1	Hardware Configuration of RTC*	12
1.2	Software Configuration of RTC*	14
2.1	Intercluster Shared Memory Models	19
3.1	The Ethernet Request Block	27
3.2	User Shared Data Queues	29
3.3	Intracluster Data Flow	31
4.1	The Ethernet Packet	35
4.2	Transmit Data Block and Receive Data Block	38
4.3	The Relation Table	41
4.4	Message Transmission	44
4.5	Message Reception	47

ACKNOWLEDGEMENT

To Karin, Svenja, and Arne in appreciation of their effort to keep the environment as stressfree as possible.

I could not have done this thesis without their patience.

I. INTRODUCTION

A. DISCUSSION

1. General

The goal of this thesis is to extend the existing version of the distributed multi-computer real time executive (E-MCORTEX) in order to provide systemwide interprocess data communication.

The existing MCORTEX version was provided by David Brewer in December 1984 [Ref. 1], and contributes to the research work done by the AEGIS Modeling Group at the Naval Postgraduate School (NPS).

The objective of this project group is research on time critical processing required by modern anti-air warfare (AAW) systems. The project group chose the AN/SPY-1A phased array radar processing unit of the AEGIS weapon system due to its challenging time critical processing demands. A further fundamental objective of the AEGIS Modeling Group is to use off the shelf components within the AEGIS weapons system as a low cost approach, and to upgrade reliability of the system by replacing the AN/UYK-7 central computer of the AN/SPY-1A system by distributed computing power. Rapid repair turnaround and low component replacement cost in the system in case of failure, and graceful degradation of the system are of utmost importance especially for military applications.

The available lab system at NPS is made up of single board computers building MULTIBUS clusters which are connected via an Ethernet Local Area Network.

2. Systems Architecture

The lab system's hardware configuration shown in Figure 1.1 consists of two clusters with up to four Intel iSBC 86/12A single board computers (SBC) in a cluster at the present time. A MULTIBUS serves as the interconnection

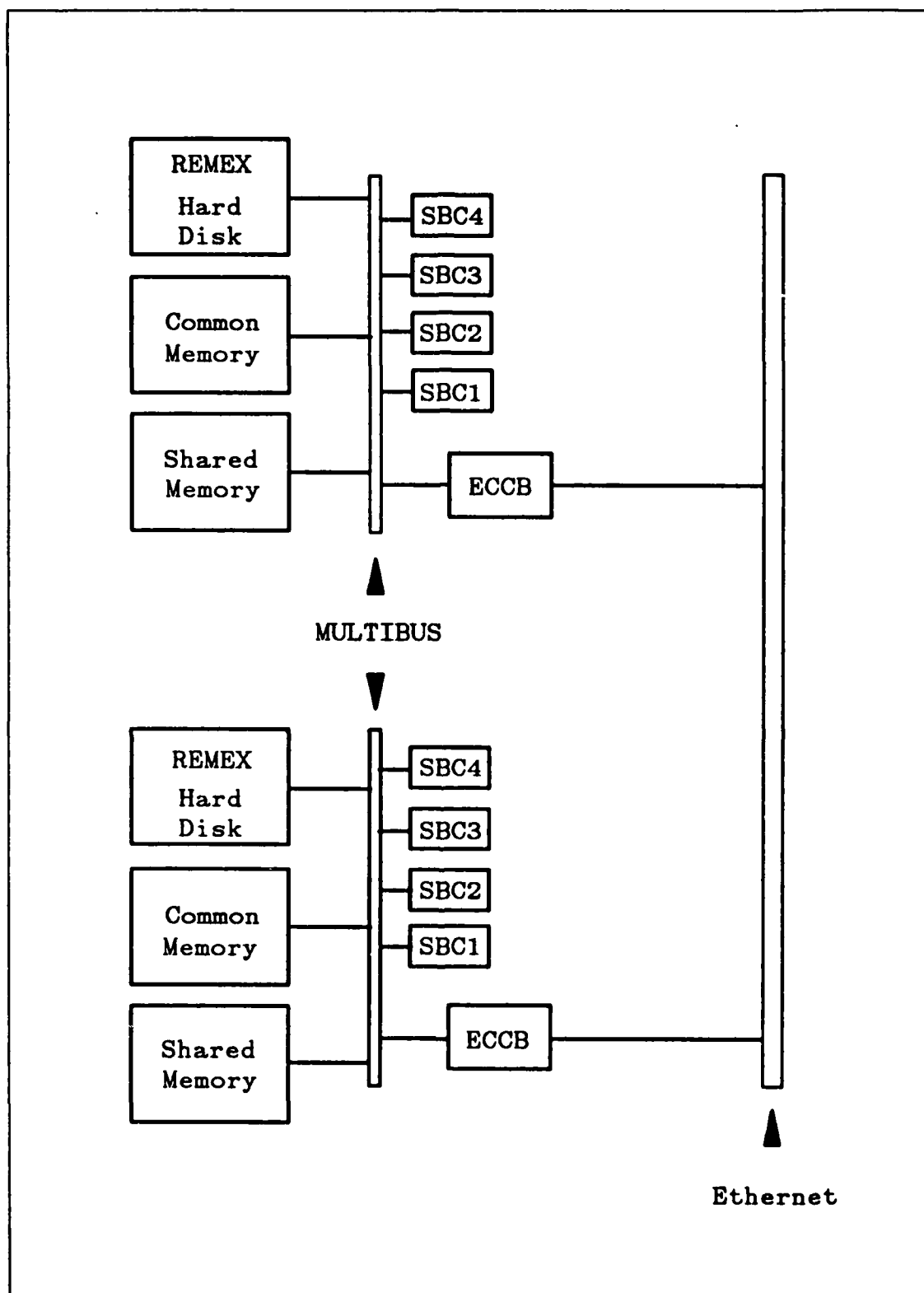


Figure 1.1 Hardware Configuration of RTC*.

medium for all cluster elements, i.e. besides the SBC's, a hard disk drive, two extra memory boards, and an interLAN NI3010 Ethernet Communication Controller Board (ECCB). The ECCB provides, via a transceiver, the cluster's connection to the Ethernet.

With the MULTIBUS as an intraccluster bus and the Ethernet as an intercluster bus the system's configuration looks similar to Carnegie Mellon's Cm* [Ref. 2]. Due to its goal to serve time critical applications in a real time environment the AEGIS lab system is known as Real-Time Cluster Star (RTC*).

The system's software configuration shown in Figure 1.2 consists of a MCORTEX kernel on every SBC, MCORTEX global data on one extra memory board, known as Common Memory, and Shared Memory on the second extra memory board in every cluster. Besides the MCORTEX kernel, also the CP/M-86 operating system and DDT-86 are available in local RAM of every SBC, and the user area provides space for application programs. The CP/M Multiuser area is kept on the Common Memory board, while Shared Memory houses user shared data and some system's shared data.

SBC 1 in every cluster is dedicated to a systems program known as the system device handler and packet processor (called a driver in the following). Part of Shared Memory is used as a data exchange buffer between any SBC in the cluster and the driver board, SBC 1, for Ethernet transmission requests. Another part serves as data exchange buffer between the driver board and the ECCB for handing over messages which are to be transmitted and messages which have been received.

The distributed MCORTEX kernels provide the multiprocessing capability of the system. System processes and user processes share the CPU of a respective SBC. David Brewer gives an in depth discussion of the interrelationship and the scheduling of processes.

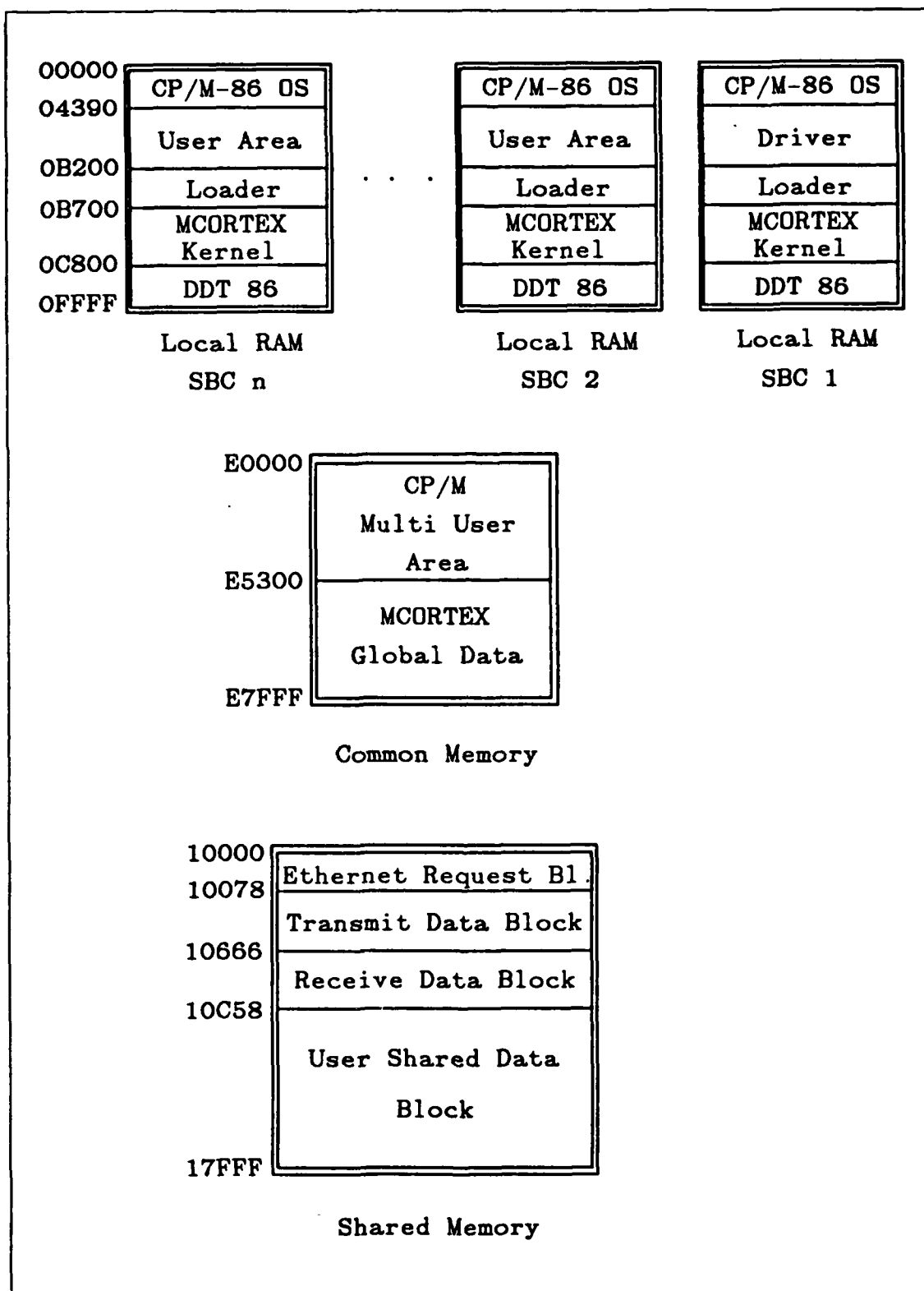


Figure 1.2 Software Configuration of RTC*.

The process synchronization is accomplished using eventcounts and sequencers as developed by Reed and Kanodia [Ref. 3]. Eventcounts synchronize all processes, those running on the same board as well as those running on different boards in the same cluster as well as those running on boards in different clusters.

It is important to recognize that eventcounts are also data, even though a special kind of data.

3. Specific

A typical application situation for systems like RTC* is time critical gathering of data by real time sensors (e.g. radar), processing these data in the context of information gathered by other sensors, and executing specific algorithms in order to produce data that are consumed by effectors (e.g. missile launchers).

Under the realistic assumption that sensors and effectors are locally distributed and that sensor-effector coupling or grouping must be kept flexible for the sake of weapon system survivability, it is obvious that the different processing modules cannot be kept together close enough in order to use shared memory in the conventional sense.

B. BACKGROUND

A series of theses starting with one by W.J Wasson, June 1980, which defined the detailed design of MCORTEX based on MULTICS and the use of eventcounts [Ref. 4], developed a highly modular system, hardwarewise and softwarewise, using commercially available components, that guarantee low cost, availability, and reliability. D.K.Rapantzikos, March 1981, provided initial implementation [Ref. 5], E.R. Cox, December 1981, refinement [Ref. 6], and S.G. Klinefelter, June 1982, dynamical interaction with the operating system during execution [Ref. 7].

W.R. Rowe, June 1984, put the multiuser CP/M-86 operating system under control of MCORTEX [Ref. 8], and

finally D.J. Brewer, December 1984, extended MCORTEX to a multicluster system without shared memory, using Ethernet as cluster interface.

The system's functioning was shown for the extended MCORTEX version up to the level of systemwide process synchronization using distributed eventcounts in a multicluster environment. Even though eventcounts are data also, and communicating eventcounts is shared data communication, this special shared data communication lacks the ability of user shared data distribution over the total system.

This thesis tackles this important step.

C. STRUCTURE OF THE THESIS

The goals of this thesis are:

1. To extend the existing MCORTEX version, that provides process synchronization and single cluster inter-systemprocess data communication using an intraccluster shared memory, to multicluster general inter-process data communication in an Ethernet Local Area Network environment.
2. To develop an appropriate model for intercluster shared memory to be used in the system.
3. To accomplish the extension without changing the MCORTEX kernel, but modifying PL/I-86 modules only.

Chapter I discusses the objective of the AEGIS Modeling Group at the Naval Postgraduate School, gives an introductory system's overview, a brief development history of the system, and outlines the goals of this thesis.

Chapter II discusses three different approaches studied in developing the concept of intercluster shared memory for RTC*, and the reasoning that lead to the decision for the chosen model.

Chapter III presents the organization of intraccluster shared memory, the use of the user shared data area, and the intraccluster data flow.

Chapter IV provides an in depth presentation of the development and realization of intercluster data sharing in RTC*.

Chapter V summarizes the current state of the system and addresses possible future enhancements.

II. ORGANIZATION OF INTERCLUSTER SHARED MEMORY

A. SHARED MEMORY MODELS

In developing the concept of intercluster shared memory for RTC*, three different approaches were studied:

- 1) no replication of intracluster shared memory,
- 2) total replication of intracluster shared memory, and
- 3) partial replication of intracluster shared memory.

The logical structures of these approaches are shown in Figure 2.1.

1. No Replication

This model views the intercluster shared memory as the sum of individual intracluster shared memories of all clusters available in the system. This is very similar to the approach chosen for Cm*.

Every shared data item is kept only once in intercluster shared memory, normally in the intracluster shared memory of the home cluster of the producing process. Consuming processes have to go through the various bus hierarchies of the system in order to access the respective item at the time when consumption is to start. Apparently this is a very time consuming effort if the respective data item is not resident close to the consuming process. Close in this context refers to being located in shared memory of the consuming process' home cluster. In terms of time efficiency for this model, the only reasonable solution for this situation is to put producing and consuming processes as close together as possible in order to increase the rate of cluster local hits.

The space-time dilemma becomes obvious. In terms of space, this approach is the most efficient one, because there exist no duplications of any data item in the whole

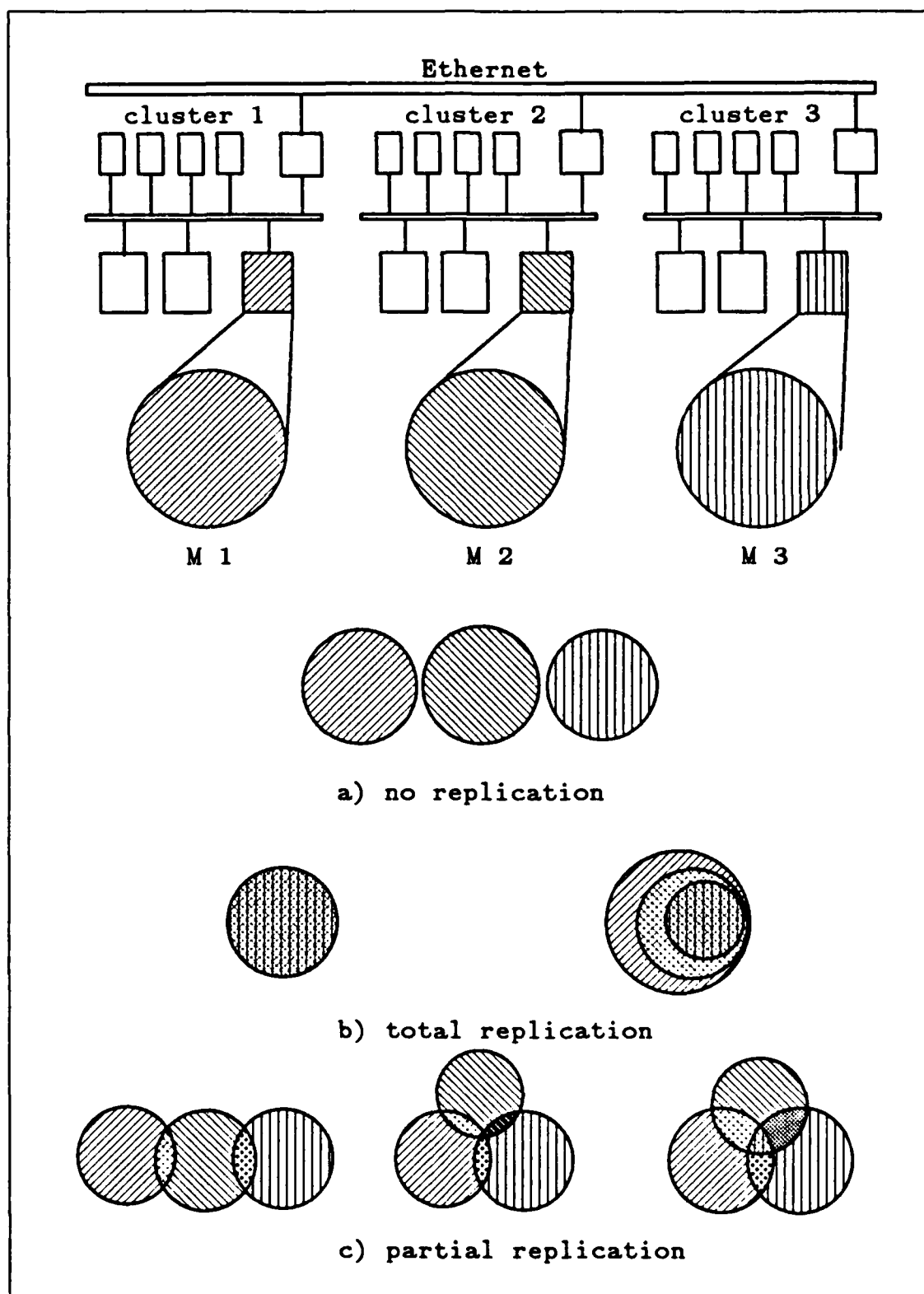


Figure 2.1 Intercluster Shared Memory Models.

system. The intercluster shared memory space is equal to the sum of all intracluster shared memory spaces in the system.

However, if we accept the fact that memory becomes cheaper and cheaper every year due to development of technologies that put more and more memory space on a chip, then we could afford to replicate data in different memories of the system in order to increase the number of cluster local hits, or even to ensure that only local hits happen in the system. This leads to the second model of intercluster shared memory.

2. Total Replication

The opposite extreme of no replication is total replication of all shared memory, i.e. every intracluster shared memory keeps all shared data items and so intercluster shared memory consists of as many copies of intracluster shared memory as there are clusters in the system. Due to the necessity that all intercluster shared memory has to fit in every intracluster shared memory, the available space for intercluster shared memory is equal to the space of the smallest intracluster shared memory of any cluster in the system. Therefore it must be ensured that the smallest intracluster shared memory is large enough to keep all shared data items needed in the system.

This model makes sure that only cluster local hits occur in the system, and that every consuming process finds every data item in shared memory of its home cluster. This seems to be a very time efficient approach that suits the demand of having data as close as possible to the producer and the consumer as well.

A major problem with this approach however is the increased overhead in maintaining and updating of shared data items that are never used at a certain cluster. This overhead especially consists of traffic on the systems buses and processing time of the hosts at the interfaces between clusters and transmission medium. This traffic overhead

slows down the data distribution so that it takes longer for data to be available at the consuming process' cluster for a local hit there.

The compromise is a combination of these first two models, where only necessary replication of data is done.

3. Partial Replication

In this model every cluster maintains shared data only if those data are used at a specific cluster, i.e. the producing and the consuming clusters only keep a copy of respective data items, no superfluous information is maintained and there is no superfluous traffic on the transmission medium and the system buses. The intercluster shared memory in this approach is equal to the union of all individual intracluster shared memories in the system and only the intersections are replicated. There can be different intersections between different cluster groups in the system. This approach is the most efficient one in terms of space and time. The traffic on the transmission medium and the system buses, and the amount of processing time needed for data exchange is kept to a minimum. Also, as in the total replication approach, only cluster local hits will occur.

The overall policy is to transmit shared data to all consuming clusters as soon as possible after production, and to transmit those data to consuming clusters only.

This approach seems to be the most adequate one for a distributed real time system, as it reflects the optimal compromise in terms of speed, space, and update overhead.

Intercluster shared memory by partial replication of intracluster shared memories was therefore chosen for implementation in RTC*.

III. ORGANIZATION OF INTRACLUSTER SHARED MEMORY

A. CLASSES OF SHARED DATA

Looking closer at shared data, we realize that there are different classes of shared data used in the system. Some data is shared among processes on the same board only, some is shared among processes on different boards in the same cluster, and some is shared among processes in different clusters.

It was decided to keep all shared data on special memory boards, even though some of the data is produced and consumed on the same board. To protect system's data used by the operating system, these data items are stored in Common Memory, which is beyond the reach of user data memory, called Shared Memory.

All MCORTEX global data are maintained in Common Memory. Every cluster has one Common Memory where all MCORTEX global data needed in the respective cluster is kept. Brewer describes the logical organization of Common Memories. Due to the fact that eventcounts are used for intraccluster and intercluster process synchronization, some of these eventcounts are replicated in more than one cluster.

It must be recognized that there exist system's eventcounts in every cluster, e.g. ERB_READ and ERB_WRITE. These are used strictly cluster internally and do not belong to the intersection of intercluster Common Memory. These system eventcount have the same name in every cluster, but their respective values are never distributed over the system. Only user eventcounts that are used in more than one cluster are distributed and therefore replicated in respective clusters.

B. SYSTEM SHARED DATA

A similar situation exists in Shared Memory. There are three data items in Shared Memory which are used exclusively within the cluster:

- 1) the Ethernet_Request_Block,
- 2) the Transmit_Data_Block, and
- 3) the Receive_Data_Block.

These are data items shared between the driver residing on SBC 1 and either MCORTEX kernels on other boards or the ECCB. These system's data items are needed to establish cluster external communication. Information needed by the driver about outgoing and incoming messages has to be communicated via Shared Memory, because it is not possible to access local memory of any SBC from outside the board.

As is true for system eventcounts in Common Memory, these three system shared data items are also used strictly cluster internally and do not belong to the intersection of intercluster Shared Memory, even though they have the same names in every cluster. Only user shared data, that are shared in more than one cluster are distributed and therefore replicated at respective clusters.

As described by Brewer, the Ethernet Request Block, the Transmit Data Block, and the Receive Data Block reside in this order in the lower part of Shared Memory at addresses 10000H, 10078H, and 10666H respectively. The User Shared Data Block starts at address 10C58H and goes up to 17FFFH as the highest address in Shared Memory in the present implementation of RTC*.

C. USER SHARED DATA

Shared data items are basically interfaces between different processes. Processes communicate via these shared data. It is therefore important to agree on the name, size, and structure of shared data used by different programmers for different process modules. This agreement must be

accepted by all programmers of any system module and can be thought of as reached under the guidance of a lead programmer.

An individual programmer can still use any other private name and structure for some variable, as long as he or she ensures that communication with any other module is done using the agreed upon name and structure.

1. Organization of Shared Data

Shared data items are organized as circular queues of structures, where the actual item is a structure and the queue serves as buffer between producers and consumers. This is true for user shared data and system shared data as well.

While system shared data items have fixed predefined structures and queue lengths, a user shared data item can be of any structure, and a user shared data queue can be of any length. The only restriction is that all user shared data queues needed at some cluster must fit together into the User Shared Data Block in Shared Memory of that cluster.

The length of specific data queues is another important issue to be agreed upon by all programmers using respective shared data items. The chosen queue length depends on the expected average input and output rate of a specific data item queue. The goal is to reduce or, if possible, to avoid idle waiting times at producing processes due to a filled up queue caused by slow consumption.

As mentioned in Chapter II, the data communicating version of RTC* developed in this thesis will use the concept of partial replication of Shared Memory. Therefore we think of intercluster Shared Memory as the union of all intracluster shared memory blocks that contain user shared data. Data communication is only possible among clusters that actually share data (i.e. there exist an intersection of intracluster Shared Memories of those clusters and the respective shared data item is in the intersection). To put it another way, an intersection of intracluster Shared

Memories is only needed if processes in respective clusters need to communicate. If intracluster Shared Memories only keep those shared data items needed (produced or consumed) by some process in the cluster, then automatically the minimum intersection and therefore the least duplicated use of memory space and the most efficient use of time for maintaining the data is guaranteed.

2. Storage of User Shared Data

In order to further accomplish efficient use of memory space, the system is set up in such a way, that user shared data queues can be stored at any address in the User Shared Data Block in Shared Memory. There must, however, be enough space available between the starting address of the queue and the highest possible physical memory address (i.e. 17FFFH in the present implementation).

This flexibility also allows for contiguous storage of all user shared data and thus the available storage space is most efficiently used.

The same data item can reside under different addresses in different clusters. The application programmers do not have to worry about the addresses, they refer to a specific data item by its name. The lead programmer will take care of assigning addresses to shared data queues. When and how this is done will be discussed in Chapter IV. For now it should suffice to realize that the organization and storage of user shared data in this implementation is done with the least duplication of data items, where each shared data is available in the local cluster.

D. RELATION BETWEEN SHARED DATA

An important relation exists between system shared data, eventcounts and user shared data, which is exploited by the driver for its data communication task.

The system internal management of a data queue is controlled using an eventcount `<dataname>_IN` and an eventcount `<dataname>_OUT`. These eventcounts have to be

distributed over the same clusters as the related data item, in order to ensure that a consumer does not read a data item before it is in the queue, and a producer does not write a new item into an already used slot before the old item has been consumed by all its consumers. The eventcount <dataname>_IN tells all consumers that an item is available, the eventcount <dataname>_OUT tells the producer that a former used slot is available for new data.

1. The Ethernet Request Block

Refer to Figure 3.1 for the following discussion. The system shared data queue, the Ethernet Request Block (ERB) is filled with Ethernet Request Packets (ERP) initiated by any process resident at the cluster when it calls for an update of the value of an eventcount that is also needed at some other cluster.

The many producers of Ethernet Request Packets are kept in sequence by the systems sequencer ERB_WRITE_REQUEST, which basically is a ticket machine that makes sure that only one packet at a time is put into the Ethernet Request Block, and that Ethernet Request Packets are put in on a first come first serve basis. The only consumer of Ethernet Request Packets is the driver on SBC 1.

An Ethernet Request Packet keeps the following information in its eight byte structure:

```
command      (i.e. 00H means eventcount)
type_name    (i.e. 8 bit eventcount id)
name_value   (i.e. 16 bit eventcount value)
remote_addr  (i.e. 16 bit addr of external cluster)
```

System eventcounts ERB_WRITE and ERB_READ play the functional roles of <dataname>_IN and <dataname>_OUT respectively for this system shared data queue.

We realize that ERPs in the ERB are in partial order. The order of packets for different eventcounts <dataname>_IN or <dataname>_OUT is of minor importance. More

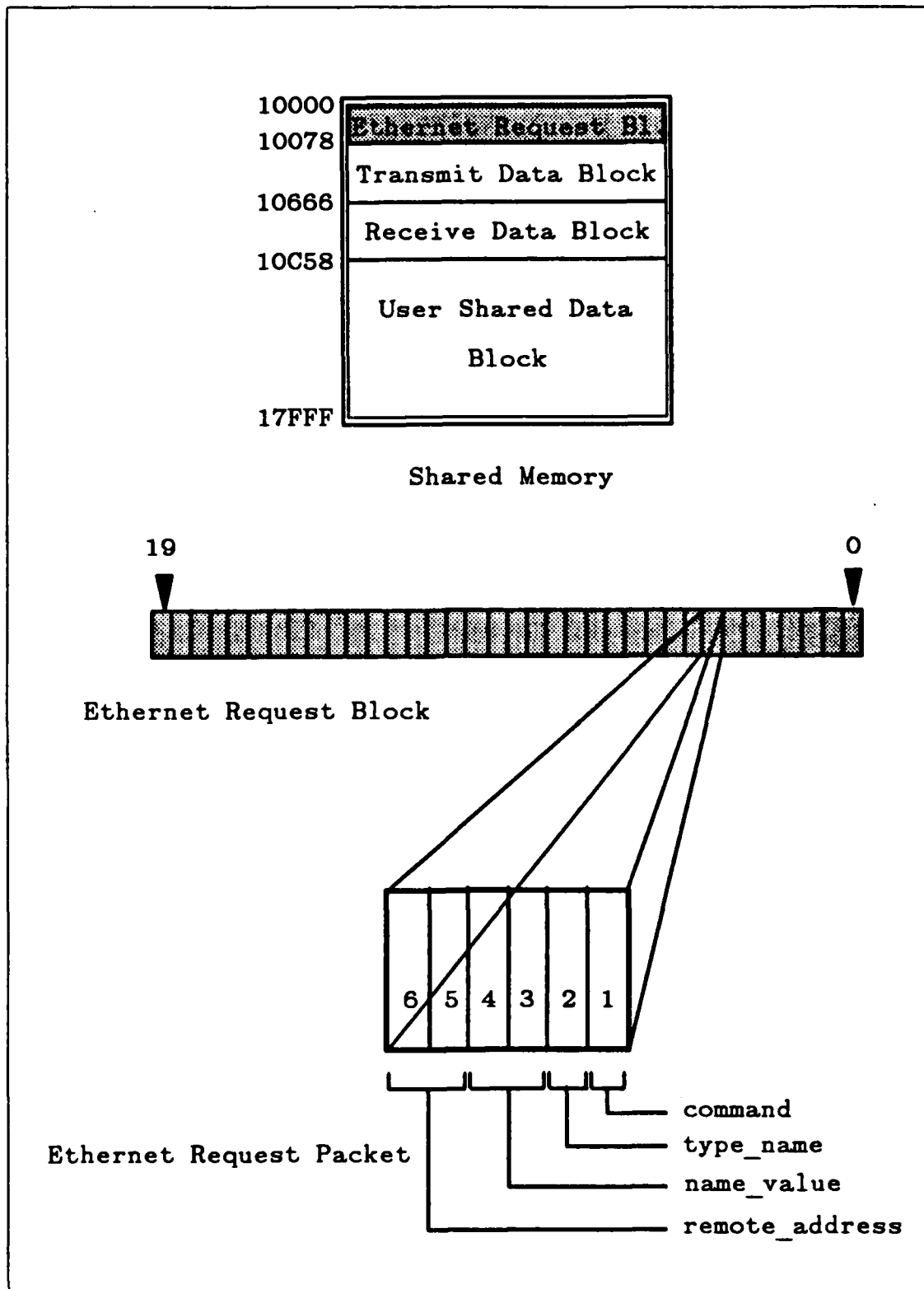


Figure 3.1 The Ethernet Request Block.

important to notice is that the logic of eventcounts, if used correctly, ensures

a) that a packet with an eventcount <dataname>_OUT always comes after a packet with the respective eventcount <dataname>_IN, and

b) that all packets with eventcounts <dataname>_IN for a specific data queue are in total order, which is also true for all packets with eventcounts <dataname>_OUT for a specific data queue.

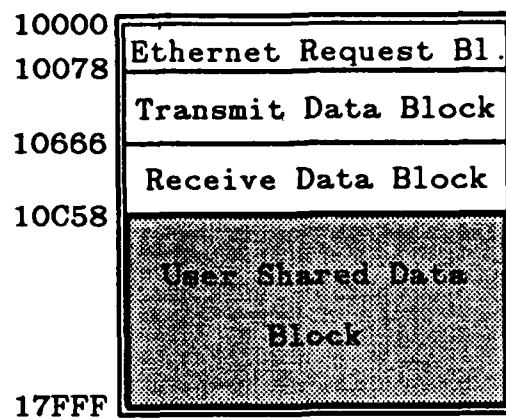
2. User Shared Data Block

As mentioned above, user shared data queues can be of any length and the data items can be of any structure, but every data item has a specific structure that is replicated in every slot of its data queue.

Due to the information contained in the eventcount value, a specific slot in a data queue has to be written before the eventcount is advanced and also the eventcount has to have been advanced before the next slot is written to. The application programmer has to be aware of this logical sequence when using eventcounts. The system then ensures that always the next higher slot in the queue is written to, and this only if this slot is available for overwrite.

This scheme also ensures that the data items in a respective queue are totally ordered. Furthermore, it is ensured that respective Ethernet Request Packets that keep value information of an eventcount related to this data item are in the same order as the data item iterations in the data queue. This is also true if an eventcount relates to multiple data.

These facts about the relation between eventcounts, Ethernet Request Packets, and user shared data is the basis for the logic of the implementation of the drivers data transmission and data reception tasks.



Shared Memory

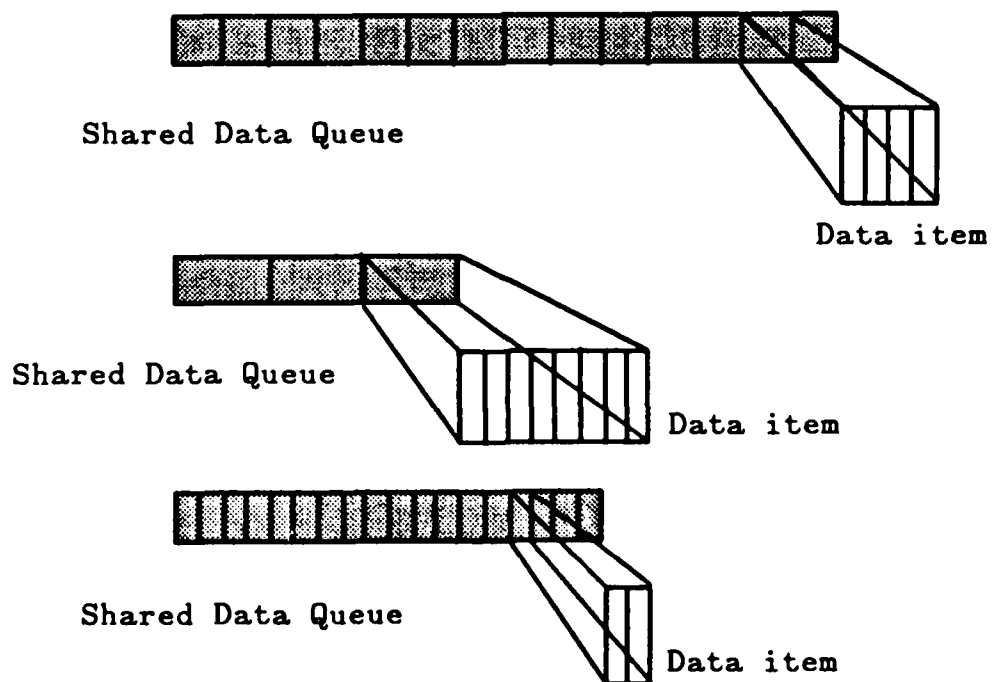


Figure 3.2 User Shared Data Queues.

E. INTRACLUSTER DATA FLOW

As mentioned earlier, we assume to have the classical producer-consumer situation for all user shared data in the system. Data items are kept in circular data queues that serve as buffers between producers and consumers. Due to this assumption, all user shared data are related to some eventcount, which replaces the semaphore used in the classical example. Refer to Figure 3.3 for the following discussion.

A user process resides in the user area of local RAM on some SBC. The process becomes active when the scheduler chooses it as the next process to run after it was ready. Before a producer process can place the produced data item into the queue, a slot must be available. Slot availability can be checked by comparing the <dataname>_IN and the <dataname>_OUT eventcount values of the respective data queue.

A consumer process becomes ready only when the value of the respective <dataname>_IN eventcount has reached the awaited threshold, indicating that there is a new iteration of the data item available in the data queue.

A process which consumes data and produces new data as well obeys the same rules. It is of utmost importance that the application programmers use eventcounts and the AWAIT and ADVANCE primitives correctly.

A producer process when running produces shared data items, puts these items into user shared memory, if there is space in the queue, and then calls the system primitive ADVANCE (EVC). This system process then increments the value of the respective eventcount and checks if this eventcount is distributed and therefore a cluster external copy needs to be updated. If so, it calls another system primitive, SYSTEM\$IO, which in turn gets a ticket from the

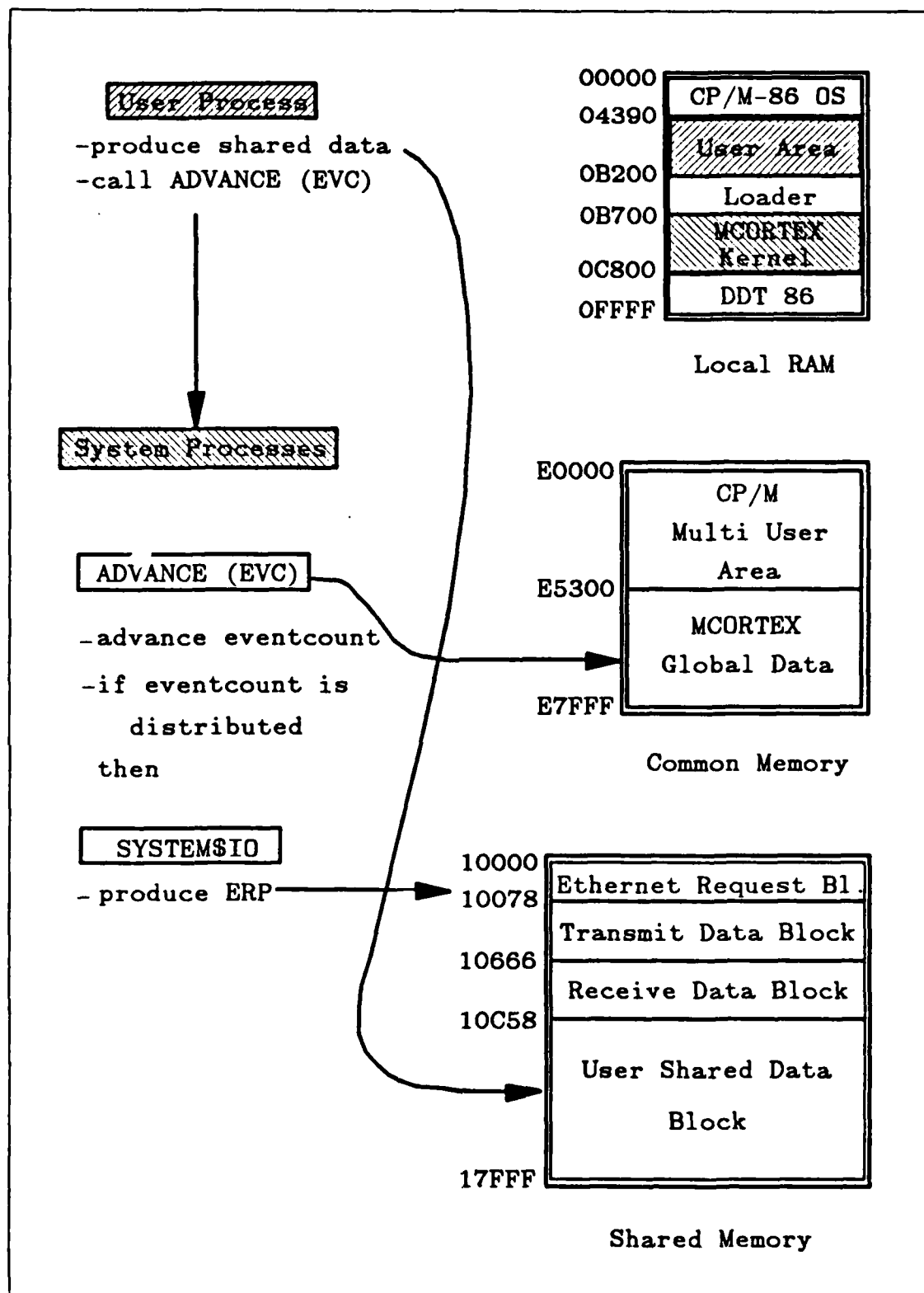


Figure 3.3 Intracluster Data Flow.

ERB_WRITE_REQUEST sequencer and puts an Ethernet Request Packet into the Ethernet Request Block when its ticket number becomes the lowest in the waiting line.

If no remote copy is needed, then no Ethernet Request Packet is produced, because all consumers reside in the same cluster as the producer, and the cluster internal synchronization can take place, as all needed data (i.e. eventcount value and shared data, if any) is present at the cluster. A waiting consumer process becomes ready and when activated consumes the shared data item from Shared Memory.

IV. INTERCLUSTER DATA SHARING IN RTC*

A. INTERCLUSTER CONNECTION

1. The Ethernet

As mentioned in the system overview in Chapter I, all clusters in RTC* are connected via an Ethernet Local Area Network. A detailed specification of the Ethernet is given by Xerox Corporation [Ref. 9]. The Ethernet provides the lowest two levels in the International Standards Organization's Open System Interconnection (ISO OSI) reference model, i.e. the Physical Layer and the Data Link Layer. Higher levels are collectively seen by the Ethernet as the Client Layer. The RTC*'s driver provides the system's Client Layer and the home board of the driver, SBC 1, serves as a host in the Ethernet's communication subnet. The physical connection of a cluster to the Ethernet's coaxial cable is provided via the ECCB NI3010, the interface between the MULTIBUS and the transceiver which actually is the tap clamped on the coaxial cable.

While the interface board provides the hardware connection between the highest level system bus (Ethernet) and the cluster bus (MULTIBUS), the ECCB software and the driver are responsible for correct exchange of messages transmitted or received by any cluster in RTC*.

As is true for all SBCs, also communication with the ECCB NI3010 has to be done via board external buffers. In contrast to inter-SBC communication, which is synchronized by eventcounts, intercommunication between SBC 1, the driver's home board, and the ECCB is synchronized using interrupts (e.g. Transmit_DMA_Done or Receive_DMA_Done).

An in depth description of the hardware NI3010 is given by InterLAN Corporation [Ref. 10]. The software driver was developed by David Brewer. Brewer's thesis provided the basic scheme for intercluster exchange of

eventcount values. The contribution of this thesis is to enhance the basic scheme for a general data exchange in the system by modifying the driver software and exploiting the logical interrelation between eventcounts and shared data.

2. The Ethernet Packet

Figure 4.1 shows the frame format of an Ethernet packet. This is a given structure produced by the ECCB. Information to be put into four of the six fields have to be provided by the client. Preamble and Frame Check Sequence are added by the ECCB for receiver synchronization and error checking respectively.

For a destination address, any six byte combination except all zeroes can be used, which is also true for the source address. The arrows on the right hand side and the bottom of Figure 4.1 indicate the serial transmission sequence of the bytes and bits. The very first bit transmitted in the destination field indicates whether the destination address is a multicast or a physical address. If this bit is 1, making the first byte of the destination an odd number, then this address is a multicast address (group address of any number of clusters). A special multicast address, all ones, is reserved as the so-called broadcast address, which addresses all participants in the network.

If the first bit is 0 then the destination is a physical address. Physical addresses are fixed addresses of ECCBs. Xerox Corporation takes care that physical addresses are unique, i.e. every physical address is used only once in any ECCB worldwide.

Due to the just described restrictions, there are actually 2^{47} different combinations available to be chosen as non-physical, non-broadcast destination addresses. For the present implementation of RTC* it was decided that two bytes are more than adequate, because the maximum number of stations for an Ethernet Local Area Network is restricted to 1024 by the Ethernet specification. Therefore the first

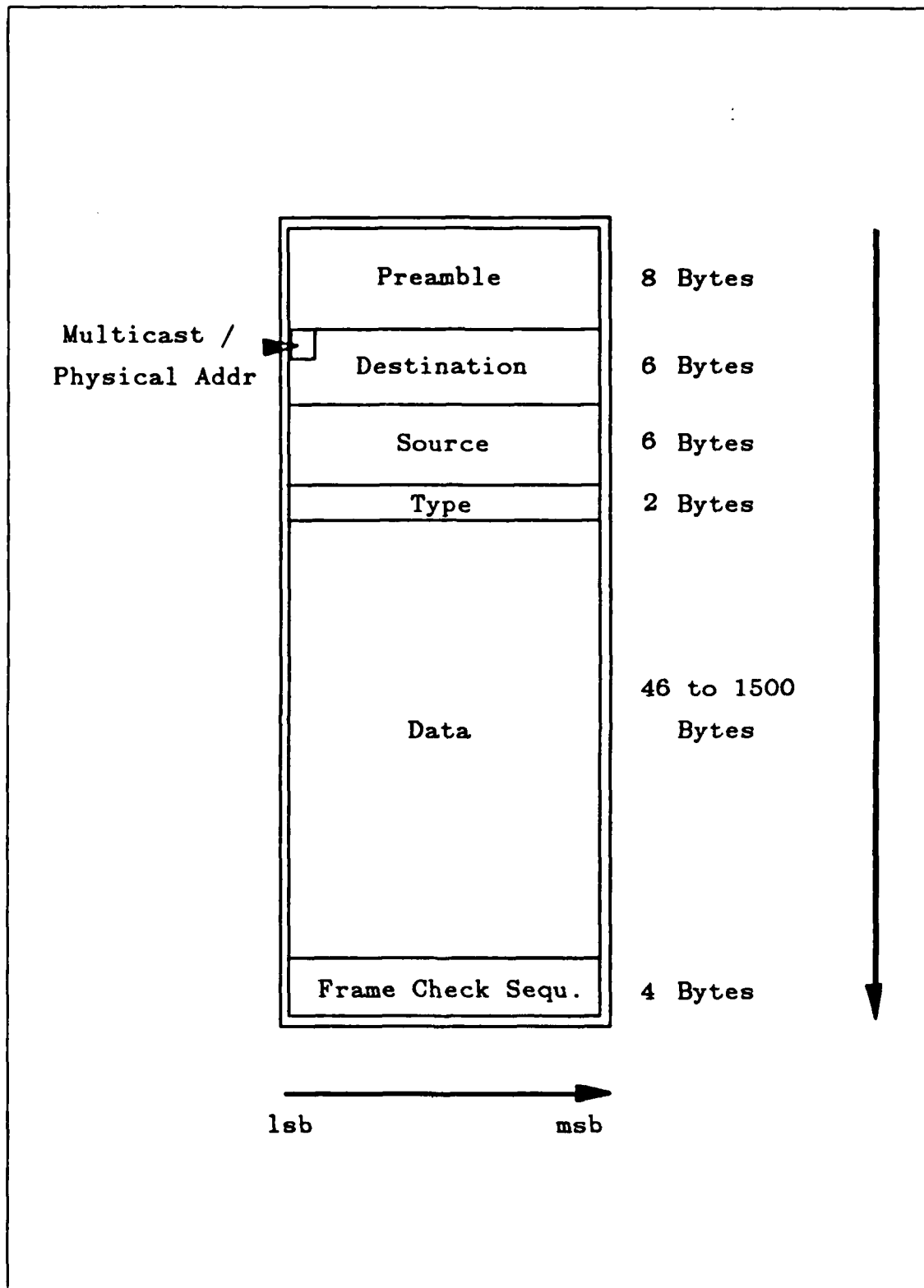


Figure 4.1 The Ethernet Packet.

four bytes of the destination are kept fixed 03H, 00H, 00H, 00H providing the multicast indication by an odd first byte.

For the the source address, the ECCB allows two possible ways. Either this address is not provided by the client, in this case the ECCB automatically inserts its physical address, or the client fills in a source address. This second way speeds up the transmission process and was therefore chosen for RTC*. The first four bytes are kept fixed 03H, 00H, 00H, 00H as in the destination field. Byte five and six contain the cluster's address. This is not the ECCB's physical address as mentioned by David Brewer, but rather a software address of the transmitting cluster.

The two bytes of the type field are reserved for use by higher levels. Clients can use this field in order to exchange information about a specific format used in the data field. The present implementation of RTC* will not use the type field and therefore sets it to 00H, 00H.

The data field provides space for up to 1500 bytes. It is required that the minimum length of the data field be 46 bytes in order to generate the minimum total length of a sufficiently long message. This minimum message size requirement guarantees that in any Ethernet collisions are detected by the sending stations, even if source and destination stations are maximum distance (2.5 km) apart, and the net performs at worst case propagation speed tolerated by the Ethernet specification. Collision detection by the sending station is important for correct backoff and retry in order not to lose messages in the network. The ECCB takes care of this minimum length requirement as will be seen later. Also the sending ECCB attaches a four byte frame check sequence at the end of every Ethernet packet in order to provide a basis for error checking to the receiving ECCB.

B. DRIVER - ECCB MESSAGE HANDOVER

By a chosen wiring option, it is not possible to access (write or read) onboard memory of an SBC or the ECCB from outside the board. Therefore a buffer for message handover is needed for outgoing messages as well as for incoming ones. These buffers are set up in Shared Memory as systems shared data Transmit Data Block and Receive Data Block. In contrast to other shared data (e.g. Ethernet Request Block or user shared data), the Transmit Data Block and the Receive Data Block are single slot queues that meet the structure requirements given by the ECCB specification described in the Ethernet Communication Controller User Manual.

1. Transmit Data Block

The Transmit Data block is a structure of 1514 bytes that contains all information required to be submitted by the client in order to enable the ECCB to build the Ethernet packet described above. Figure 4.2 shows this structure. The destination and source fields are filled with the preset address parts as well as the dynamically changing two high bytes of the destination. The type field keeps the values 00H, 00H, and the actual message content is kept in the lower part of the 1500 bytes data field.

2. Receive Data Block

The Receive Data Block, see Figure 4.2, is similar to the Transmit Data Block and carries all information contained in an incoming message, i.e. destination, source, type and data field. In addition, the receiving ECCB hands over status information related to the message, that can be used by the client in order to determine the length and the error status of the received packet. These additional items of information are kept in the first four bytes (1 byte frame status, 1 null byte, 2 bytes frame length) and the last four bytes (frame check sequence) of the Receive Data Block, making the Receive Data Block 1522 bytes long.

Destination (A)
(B)
(C)
(D)
(E)
(F)
Source (A)
(B)
(C)
(D)
(E)
(F)
Type Field (A)
(B)
Data (first byte)
.
.
.
Data (last byte)

Frame Status
0
Frame Length <7:0>
Frame Length <15:8>
Destination (A)
(B)
(C)
(D)
(E)
(F)
Source (A)
(B)
(C)
(D)
(E)
(F)
Type Field (A)
(B)
Data (first byte)
.
.
.
Data (last byte)
CRC <24:31>
CRC <16:23>
CRC <08:15>
CRC <00:07>

Figure 4.2 Transmit Data Block and Receive Data Block.

The present implementation of RTC* ignores the frame check sequence and concerns itself only with the data field.

Even though the length of the data field in the Ethernet packet is determined by the length of the actual message, both, the Transmit Data Block and the Receive Data Block provide space for maximum length messages in order to be prepared for any legal message size.

C. MESSAGE TRANSMISSION AND RECEPTION

The main task of the driver on SBC 1 is to build a message that is to be transmitted over the Ethernet, and to process a message that was received. For transmission, the message has to be built in the Transmit Data Block in Shared Memory first, and then the ECCB is to be triggered for transmission. For reception, after the ECCB signaled that it has put a message into the Receive Data Block in Shared Memory, the correct data queues in Shared Memory have to be found and the data items have to be put into the correct slots in their respective data queues.

In order to be able to do this correctly, every driver maintains a table in its local RAM which contains all the information needed about the relationship between eventcounts and shared data items.

1. The Relation Table

We assume that every user shared data item used in the system is related to some eventcount. The driver exploits <dataname>_IN eventcount, because this is the trigger for shared data transmission when put into an Ethernet Request Packet by the SYSTEM\$IO process.

Only if a <dataname>_IN eventcount was advanced, was there a new data item put into the respective data queue. An eventcount can also serve as a <dataname>_IN indicator for multiple data items that are updated at the same time. The important property is that for every shared data item there exist one <dataname>_IN eventcount, which is advanced after

the data item is available in its corresponding data queue in Shared Memory. Only if an eventcount is distributed over the system, and therefore a cluster external copy is needed, is there an Ethernet Request Packet produced and put into the Ethernet Request Block. Every data item is distributed over the same clusters as the eventcount to which it is related. A `<dataname>_OUT` eventcount only informs everybody in the system that a slot in the respective data queues becomes available for overwriting (an important item of information for producer processes), and so these eventcounts, if distributed, will be transmitted alone, with no user shared data in the same Ethernet message.

This logical interrelation between eventcounts and user shared data is kept in the relation table shown in Figure 4.3, which is a structure of 100 entries (the present implementation of RTC* allows for 100 eventcounts per cluster), which holds the `eventcount_id` and the number of related data items on level two, and information about up to 10 data items for every eventcount on level three. Besides the knowledge about how many data items belong to some eventcount, the driver needs to know, where to find a data item, what is the items structure size, what is the data queue length, what is the next slot to be sent, and what is the next slot into which to put a received item.

The relation table is built by the driver during system initialization by the procedure `make_table`, see Appendix A. This procedure reads the file `relation.dat`, which has to be present on the disk that keeps the cluster's software.

The driver (see Appendix D) is a general system process that is identical at every cluster. The relation table is cluster specific and keeps cluster specific information only. The `relation.dat` file has to be set up by the lead programmer, who decides what application program is

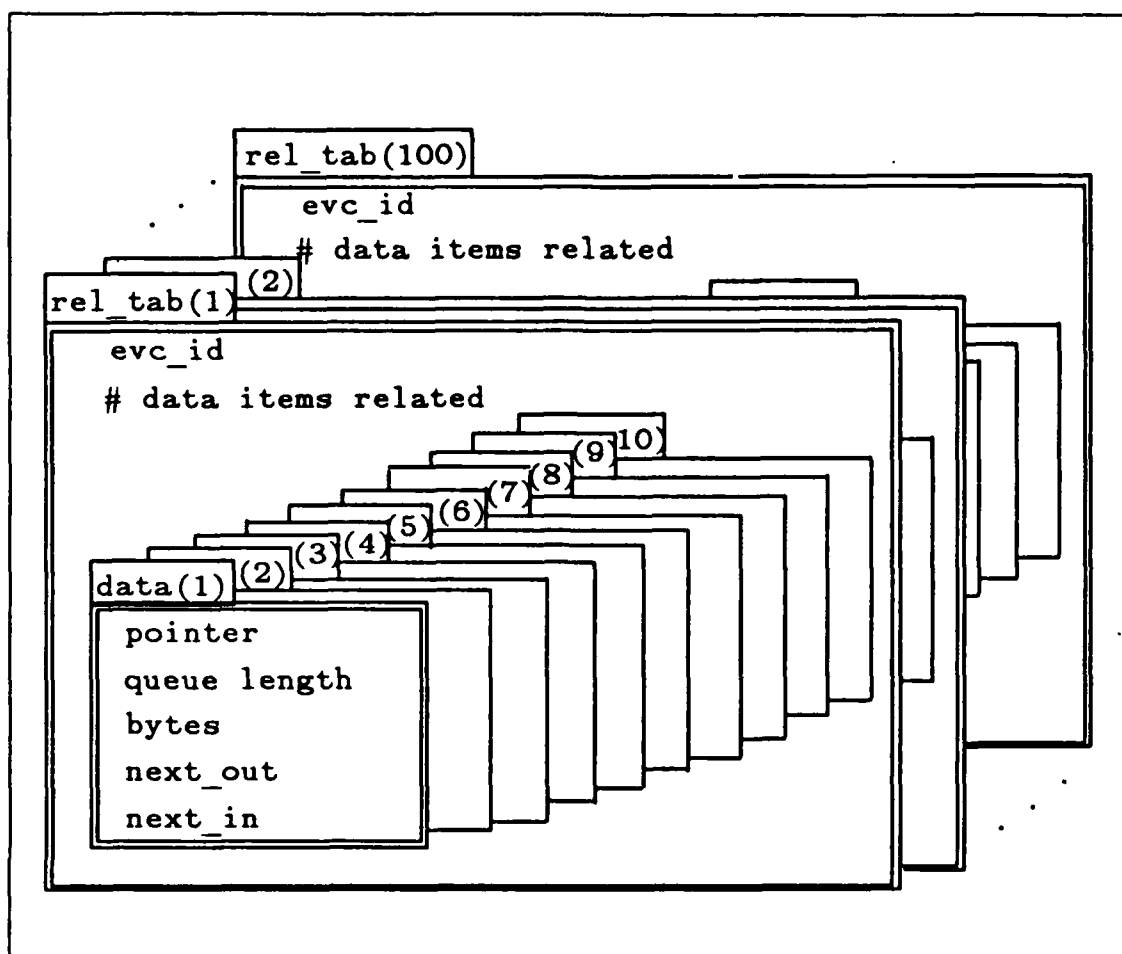


Figure 4.3 The Relation Table.

to be run at what cluster, and therefore knows what eventcounts and user shared data are needed at a cluster.

The relation.dat file basically is a table consisting of five columns as shown in Table I. It keeps eventcount identification, number of data related to this eventcount, and for every data item the address of the first byte in its data queue, the length of the data queue (# of slots), and the length of the data item structure(# of bytes). The last line in the relation.dat file serves as a sentinel consisting of zeroes in all five columns. The procedure make_table expects the formatted indata to be in columns 5, 15, 25, 35, 45 respectively. This information is

TABLE I
FILE RELATION.DAT

evc_id	numdat	point	qlen	bytes
col 5	col 15	col 25	col 35	col 45
01	1	8c58	50	6
02	3	8d84	10	3
		8da2	20	5
		8e06	2	10
05	2	8e1a	30	5
		8eb0	50	5
07	1	9072	15	5
04	1	90bd	20	8
00	0	0000	00	00

read into the relation table entries evc_id, numdat, pointer, qlen, and bytes respectively. Next_in and next_out are initially 0 and therefore do not have to be read in.

The information kept in the relation table suffices for the driver to do its job. The driver does not need to know anything about the logical structures or names of data items. It treats a data item as a sequence of bytes, and is only interested in finding the correct sequence of bytes for transmission, or the correct place in Shared Memory to put the bytes after reception. Procedures make_message and process_packet take care of this.

2. Data Format

A decision that had to be made was in what manner the data field of an Ethernet Packet should be used in order to exchange data in RTC*. The question was discussed whether to use different fixed formats for different situations and using the type field for identification of the format used in the data field. After recognizing the logical interrelationship between eventcount and shared data, that carries the possibility of uniquely identifying a group of shared data by its common eventcount, it was decided to keep

the data format as flexible as possible.

The first four bytes of the data field will always keep the eventcount information followed by as many data items as are related to this eventcount. The maximum length of these data items together is restricted to 1496 bytes in order to respect the 1500 byte limit of the data field when the 4 byte eventcount information is included. This seems to be more than adequate for the purpose of RTC*, and still leaves the possibility to transmit all data items as long as a single item is not longer than 1496 bytes, by logically grouping data items under eventcounts respecting this restriction.

The eventcount is the identifying part, therefore only one eventcount is transmitted in any Ethernet packet.

3. Message Transmission

Message transmission is triggered by an Ethernet Request Packet (ERP) available in the Ethernet Request Block (ERB); more precisely, by an advanced eventcount ERB_WRITE indicating that there is an ERP in the ERB which has not been processed yet. The driver with its preference for outbound messages will start a transmit job as soon as possible. In the initialization part, the driver already has preset the first four bytes of the destination field and all six bytes of the source field. Refer to Figure 4.4 for the following discussion.

Bytes five and six of the ERP are copied into the two high bytes of the destination field of the Transmit Data Block, making the first 14 bytes of the Transmit Data Block complete.

Next a four byte overlay is put over the ERP, using a based variable [Ref. 11], after which the procedure make_message (see Appendix B) is called.

This procedure first checks if the ERP contains an eventcount (in the present implementation only eventcount related ERPs are processed). If byte one of the ERP

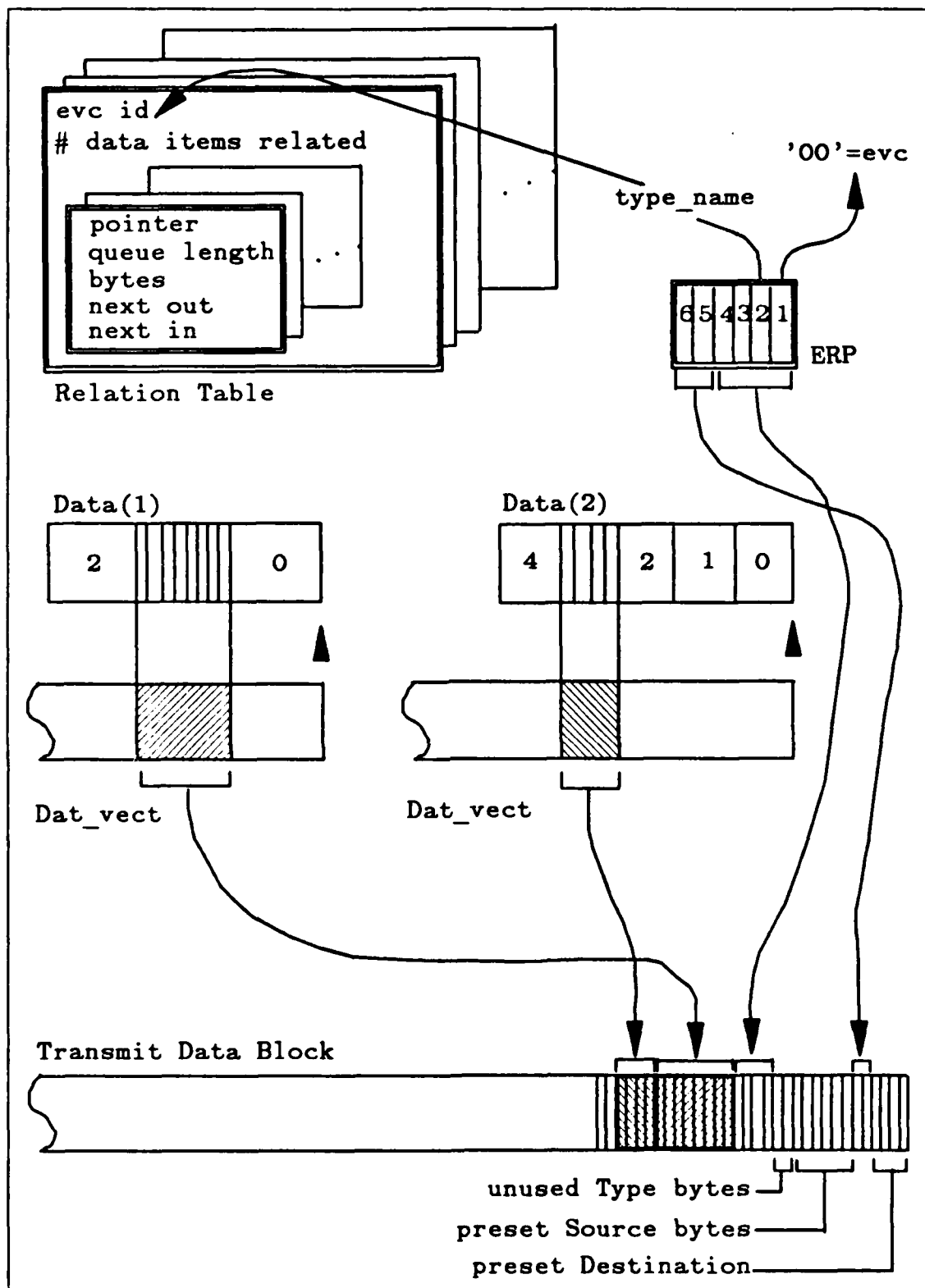


Figure 4.4 Message Transmission.

contains 00H, indicating EVC_TYPE then the first four bytes of the ERP are copied over into the first four bytes of the data field of the Transmit Data Block.

Next a relation table look up is done under the respective eventcount_id and the number of related data items is found. If the eventcount_id is not in the table, then there are no related data and the message is done, otherwise the first related data item is found, an overlay (1500 bytes in the present implementation) is aligned with the data queue, using the address information (pointer) of the data item. Now the slot number (next_out) and item size(bytes) are combined to an offset in order to find the first byte to be copied over into the Transmit Data Block to follow the eventcount information in the data field. The data item size (bytes) contains the number of bytes to be copied over.

The next_out of this data item in the relation table is updated to the next slot number, and the loop starts again for the next data item related to this eventcount. Meanwhile also the total bytecount for bytes put into the data field of the Transmit Data Block is carried on. After all the related data has been copied over into the Transmit Data Block, this bytecount information is added to 14 (6 bytes destination, 6 bytes source, 2 bytes type) and the resulting bytecount is used as a parameter in the procedure transmit_packet, which signals the ECCB that a message is ready to go and should be sent.

Just before calling procedure transmit_packet, the driver calls ADVANCE (ERB_READ), which makes the just processed ERP slot available for reuse.

The ECCB copies the number of bytes signaled (minimum 60) into its transmit queue and puts the message out over the Ethernet. If necessary due to collisions, the transmission is repeated and only after the message was successfully sent does the transmitter become ready for the

next transmission, which is prepared by the driver in the above described fashion.

4. Message Reception

Message reception is triggered by an ECCB interrupt signaling that there is a received message available in the ECCB's receive queue. Refer to Figure 4.5. The driver then initializes a DMA and the ECCB puts the message into the Receive Data Block in Shared Memory. After the message is in the Receive Data Block, the procedure `process_packet` (see Appendix C), is called. This procedure works similarly to the procedure `make_message`.

Instead of getting the needed information from an ERP, procedure `process_packet` has to look up the first bytes in the data field of the received message. This implementation of RTC* neither uses the frame status and frame length information, nor the frame check sequence.

First the procedure `process_packet` looks up byte one of the data field in order to check if the just received message contains eventcount information. If so, it finds out if the value of the just received eventcount is higher than the local value of the respective eventcount, because the message is of interest for this cluster only if the remote eventcount value is more advanced than the local one.

If the remote value is higher, a relation table look up is made under the `eventcount_id` found in byte two of the data field.

If there is no entry in the relation table for this eventcount, then no related data exists and the only thing to do is to update the local eventcount value.

If an entry exists, then the address (pointer) of the first related data item is found, an overlay is aligned with the respective data queue, and the slot number (`next_in`) and the item size (bytes) are combined to an offset in order to find the first byte in the data queue that is to be changed. Then the number of bytes found in the

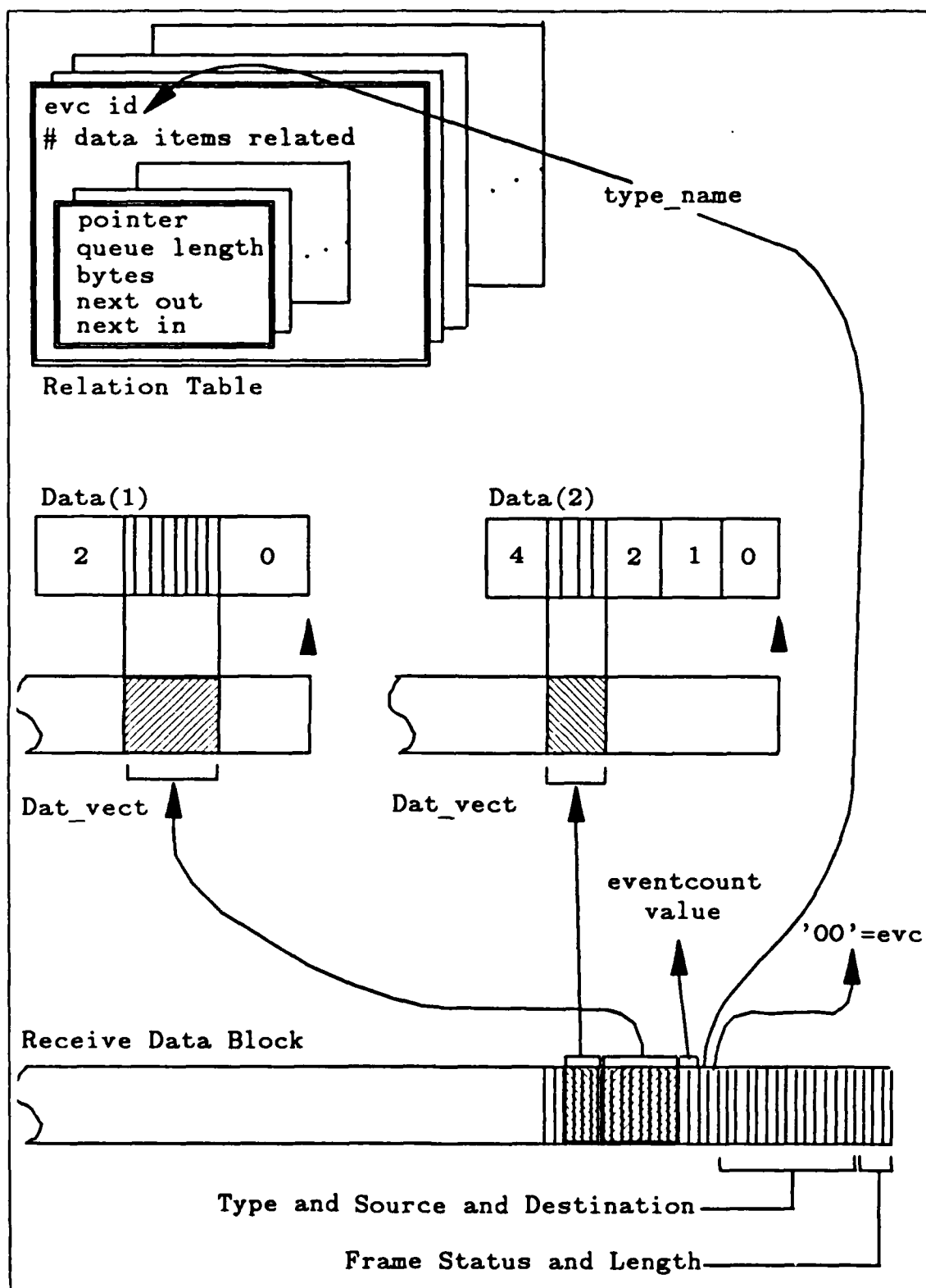


Figure 4.5 Message Reception.

item size information is copied over into the data queue, starting with byte five of the Receive Data Block's data field (i.e. the first byte of the first related data item received).

During this operation also the bytecount is updated in order to find the starting byte for the next related data item.

Similar to the procedure `make_message`, the procedure `process_packet` updates the `next_in` information to the next slot number in order to be ready for the next incoming message bringing an update for this data item if any.

After the first shared data item is copied into its correct slot in Shared Memory, the next related item is copied into its respective queue. After all received related data items of the received eventcount are updated, the eventcount is advanced to its new value, signaling the shared data status to respective consumers at this cluster. The procedure `process_packet` takes care of advancing the eventcount only after all related shared data items are updated, guaranteeing the consistency of eventcount values and data items.

D. DATA SHARING

It is obvious that data sharing using buffers in Shared Memory can only be achieved when producers and consumers agree upon, where to put and to find the respective data items. Also, this only works, if producer and consumer deal with the same item structure.

In a system like RTC*, where probably many applications programmers write different system modules, these programmers have to agree upon the shared data names and the structures and queue lengths (at least for intracluster sharing).

Even though it would suffice to only declare those shared data items that are actually used in some process, the policy followed in the demonstration program was to

include a common declaration file in every module in order to ensure that sharing modules really work with the same data item. Following this example in a real program makes it easier to maintain all shared data declarations, probably done by the lead programmer.

Applications programmers include the shared data file in their programs and only have to be concerned about the correct use of those items actually used in their programs. Table II shows the file share.dcl for the demonstration program.

TABLE II
FILE SHARE.DCL

```
DECLARE
    (de_ptr,tr_ptr,mo_ptr) pointer,
    1 delta(0:19) based(de_ptr),
      2 dx fixed bin (7),
      2 dy fixed bin (7),
      2 dz fixed bin (7),
    1 track(0:49) based(tr_ptr),
      2 x fixed bin (15),
      2 y fixed bin (15),
      2 z fixed bin (15),
    1 missile_order(0:49) based(mo_ptr),
      2 launcher fixed bin (7),
      2 azimuth float binary,
      2 elevation float binary;
```

This file ensures unique declarations for all user shared data in the whole system.

Every user shared data item is declared as a queue that is based on a respective pointer. Using based variables provides the possibility that in spite of total user shared data declaration, only for those items that are to be resident in some cluster's Shared Memory physical memory space is assigned. This leads to efficient use of memory

space. As mentioned before, contiguous storage of data queues enhances the efficiency even more.

This requires thoughtful assignment of addresses to the different data queues in the system. As for the relation.dat file and share.dcl file, the custodian for the assignment of pointers also should be the lead programmer. Applications programmers do not have to worry about this because they refer to a data item by dataname. Pointer assignments are kept in the file pointer.ass, which is cluster specific; the share.dcl file is the same for every cluster in the system. Table III shows the two pointer.ass files used in the demonstration program.

TABLE III
FILE POINTER.ASS

```
/*
  this file keeps the pointer assignments
  for shared variables used at cluster 1 */
unspec(tr_ptr)='8c58'b4;
unspec(mo_ptr)='8d84'b4;

/*
  this file keeps the pointer assignments
  for shared variables used at cluster 2 */
unspec(tr_ptr)='8c58'b4;
unspec(de_ptr)='8d84'b4;
unspec(mo_ptr)='8dcd'b4;
```

In order for processes to be able to really share data in Shared Memory, it is important that they find shared data under the same physical address. Under INTEL's policy that calculates a 20 bit physical address from a segment and an offset, this implies that user shared data has to be found in the same segment, the pointer or logical address then is the offset in this segment.

In RTC* this is realized in using 800H as a data segment register value and using sixteen bit pointers for shared data starting at 8000H. The lowest Shared Memory address therefore is $800H * 10H + 8000H$, which is equal to $8000H + 8000H$, or 10000H. The lowest byte of the Ethernet Request Block resides at the above address.

The segment used by a process is defined in procedure `create_proc`. It is important that the parameters 4, 7, and 8, i.e. stack segment (SS), data segment (DS), and extra segment (ES) in the `create_proc` call are set to 800H when creating a process. As mentioned by Brewer, when he describes user process creation [Ref. 1: p. 49], some PL/I-86 routines assume identical contents in the SS, DS, and ES registers.

V. CONCLUSION

The goals of this thesis were achieved. The MCORTEX real time executive is extended to handle multicluster general inter-process data communication. An appropriate model for intercluster shared memory is implemented by partial replication of intracluster shared memory. Only PL/I-86 modules were modified or newly added.

The message exchange scheme is kept as flexible as possible, with the only restriction that the four bytes of eventcount information have to be put into the first four bytes of the data field of the Transmit Data Block, and all related data items have to follow in the sequence given by the relation table. The driver takes care of this.

Maximum data length in a single message is restricted to 1500 bytes in accordance with the Ethernet specification. This size seems more than adequate for the purpose of RTC*. If longer messages are needed in the system, a correct data exchange can be achieved by breaking up the message into smaller ones relating these to specific eventcounts.

The driver takes care of correct message assembling and processing, and is -- as a special systems module with a dedicated board -- completely transparent to the applications programmer and user. The lead programmer will have to decide how to distribute different applications modules, and where to store data queues in Shared Memory. He or she will have to maintain the relation.dat file and the pointer.ass file, and also the agreed upon user shared data in the share.dcl file.

In the current implementation of RTC* the distributivity of the eventcounts (and with these the distributivity of data items) have to be set at system initialization. This restricts the dynamic reconfiguration of the system after initialization. Future implementations should try to resolve

this restriction. A possible way might be to exploit the general broadcast situation of an Ethernet environment. As only one message can be on the Ethernet at a time, and as all stations on the net have to listen and cannot do anything else during this time, this situation could be exploited in the following manner.

Use the eventcount_id as a kind of multicast address. As there is only one eventcount in any one message this is a unique identification of what information is carried in the message. Every cluster "knows" what information is needed at that cluster. If the eventcount_ids of related data items needed at some cluster are put into the group address table of that cluster's ECCB, then every Ethernet packet that carries information of interest for this cluster will be taken in and processed.

It is not necessary to keep the remote address for an eventcount in Common Memory. The information that an eventcount is distributed or not distributed, meaning a cluster external copy is needed or not needed, suffices. This distributivity information can be initialized for the initial system constellation. On reconfiguration, it could then be automatically and dynamically changed without having to shut down and reinitialize the whole system.

After reconfiguration, only those clusters where actual changes were made broadcast the eventcount_ids of interest to the cluster. Every other cluster updates its distributivity information for those eventcounts.

There was not enough time for the above described implementation in this thesis, but future work in this direction is highly recommended in order to make the total system more efficient, more robust, more survivable, and more flexible, requirements that are of utmost importance especially for military applications.

APPENDIX A
PROCEDURE MAKE_TABLE

Procedure make_table is the first procedure called by the driver. It sets up the relation table in local RAM of SBC 1 by reading the information from the file, relation.dat.

The relation table is a three level structure that keeps the eventcount_id (evc_id) and the number of data items (numdat) related to this eventcount on level two and the data queue address (point), number of slots in dataqueue (qlen), number of bytes in item structure (bytes), next slot to be sent (next_out), and next slot to be received (next_in) on level three.

There are maximum 100 level one entries in the relation table, because the maximum number of eventcounts at any cluster in the present implementation is 100. For every eventcount a maximum of 10 related data items are possible. The driver looks up an eventcount_id and finds all information necessary to either combine data items in the Transmit Data Block for transmission, or put received data items in their respective Shared Memory slots.

Procedure make_table expects the indata evc_id at column 5, numdat at column 15, point at column 25, qlen at column 35, and bytes at column 45 in the relation.dat file. Next-out and next_in are initially 0 and do not have to be read.

```

*****
*****
***   PROCEDURE MAKE_TABLE           R.Haezger, Dec 1985   ***
***-----***
*** This procedure reads relation values from file      ***
*** RELATION.DAT into the relation table.              ***
*****

```

```

make_table: procedure ;

declare
    relation file,
    (j,i) fixed bin (15),
    eof bit(8);

open file (relation) stream input;
i=0;
eof='0'b4;
do while (eof='0'b4) ;
    i=i+1;
    /* read data from relation.dat file */
    get file (relation) edit(rel_tab(i).evc_id,
                             rel_tab(i).numdat)
                             (column'5',b4(2),column'15',f(2));

    do j=1 to (rel_tab(i).numdat);
        /* read data for all related items */
        get file (relation) edit
            (unspec(rel_tab(i).data(j).point),
             rel_tab(i).data(j).olen,
             rel_tab(i).data(j).bytes)
            (column(25),b4(4),column(35),f(4),column(45),f(4));

        maxbytes=max(maxbytes,
            rel_tab(i).data(j).bytes*rel_tab(i).data(j).olen);

    end;
    /* if sentinel is reached */
    if rel_tab(i).evc_id = '00'b4 then
    do;
        eof='1'b4;
        put skip list('longest data queue at this cluster:',
            maxbytes,' bytes');
    end;
end;
end make_table;

```

```

*****

```

APPENDIX B
PROCEDURE MAKE_MESSAGE

Procedure make_message is called by the driver when there is an Ethernet Request Packet in the Ethernet Request Block that has not been processed yet.

It checks the ERP for eventcount type, and if the ERP contains eventcount information, it sets up the data field of the Transmit Data Block. The eventcount information is always put into the first four bytes of the data field, followed by all data items related to this eventcount in the sequence given by the order of these data items in the relation table.

Procedure make_message also keeps track of the bytecount of the total message, an information needed by the ECCB for transmission.

```

*****
*****
***      PROCEDURE MAKE_MESSAGE      R. Haeger,  Dec 1985 ***
***-----**
*** This procedure builds the Transmit Data Block for      ***
*** a message to be transmitted over the Ethernet.        ***
*****

```

```
make_message: procedure ;
```

```
declare
```

```

    dat_ptr pointer,
    dat_vect(1500) bit (8) based (dat_ptr),
    (next_out,r,j) fixed bin (7),
    (off,start,last,k) fixed bin (15);

```

```
/* check for evc */
```

```
if erp_vect(1)=EVC_TYPE then
do:
```

```
    bytecount=4;
```

```
    do k=1 to 4;
```

```
        /* put evc info into data field */
```

```
        transmit_data_block.data(k)=erp_vect(k);
```

```
    end;
```

```
/* check message
```

```
put skip list('::::',transmit_data_block.data(1),':::',
               transmit_data_block.data(2),':::',
               transmit_data_block.data(3),':::',
               transmit_data_block.data(4),'::::');
*/
```

```
r=1;
```

```
/* find respective relation table entry */
```

```
do while ((rel_tab(r).evc_id ^= erp_vect(2)) &
          (rel_tab(r).evc_id ^= '00'b4));
```

```
    r=r+1;
```

```
end;
```

```
/* if evc entry */
```

```
if rel_tab(r).evc_id ^= '00'b4 then
do:
```

```
    /* for every related item */
```

```
do j=1 to rel_tab(r).numdat;
```

```
    start=bytecount + 1; /* find where to put */
```

```
    last=rel_tab(r).data(j).bytes; /* how long it is */
```

```
    bytecount=bytecount + last; /* keep track */
```

```
    dat_ptr=rel_tab(r).data(j).point; /*align datvect*/
```

```
    next_out=rel_tab(r).data(j).next_out;
```

```
    /* compute offset of item in data queue */
```

```
    off=(rel_tab(r).data(j).bytes * next_out) + 1;
```

```

        /* compute next slot number to go */
        rel_tab(r).data(j).next_out=mod(next_out + 1,
                                         rel_tab(r).data(j).olen);
        do k=0 to (last-1);
            /* put item's bytes into data field */
            transmit_data_block.data(k+start)=dat_vect(k+off);
        end;
    end;
end;

        /* compute total bytecount for message */
        bytecount=bytecount + 14;
    end;
else do;    /* if not evc */
    end;
end make_message;

```

APPENDIX C
PROCEDURE PROCESS_PACKET

Procedure process_packet is called by the driver when there is a newly received message in the Receive Data Block.

It checks the received data for eventcount type, and if the message contains eventcount information, it checks if the received remote eventcount value is higher than the local value of the respective eventcount. Only if the remote value is higher than the local value, it processes the received message. Using the eventcount_id in byte two of the data field, a relation table look up is done and all received data items are put into their correct slots in Shared Memory.

After all data items are placed correctly, the local copy of the respective eventcount is updated by calling ADVANCE(EVC).

```

*****
*****
***      PROCEDURE PROCESS_PACKET      R.Haeger, Dec 1985  ***
***-----
*** This procedure processes received messages.          ***
*** It takes the data from the Receive Data Block and    ***
*** put every item in its correct slot in Shared Memory. ***
*** It also calls for an update of the evencount value.  ***
*****

```

```
process_packet: procedure;
```

```
DECLARE
```

```

    evcid bit (8),
    local_evc_value bit (16),
    (dat_ptr,p) pointer,
    remote_evc_value bit (16) based (p),
    dat_vect(1500) bit (8) based (dat_ptr),
    (off,start,last,k) fixed bin (15),
    (next_in,r,j) fixed bin (7);
put skip list('receiving');
/* check for evc */
if receive_data_block.data(1)=evc_type then
do;
    p = addr(receive_data_block.data(3));
    evcid=receive_data_block.data(2);
    local_evc_value = read(evcid);
    if local_evc_value < remote_evc_value then
    do;
        r=1;

        /* find evc entry in relation table */
        do while ((rel_tab(r).evc_id ^= evcid) &
                    (rel_tab(r).evc_id ^= '00'b4));
            r=r+1;
        end;
        if rel_tab(r).evc_id ^= '00'b4 then
        do;
            bytecount=4; /* jump over evc info */

            do j=1 to rel_tab(r).numdat;
                start=bytecount+1; /* compute start of item */
                last=rel_tab(r).data(j).bytes; /* and length */
                bytecount=bytecount+last; /* and item's end */
                next_in=rel_tab(r).data(j).next_in;
                /* compute offset in data queue */
                off=(last*next_in) + 1;
                /* compute next slot number to fill */
                rel_tab(r).data(j).next_in=mod(next_in+1,
                                                rel_tab(r).data(j).alen);
            end;
        end;
    end;
end;

```

```

        dat_ptr=rel_tab(r).data(j).point; /* align datvect*/

        do k=0 to (last-1);

            /* put item bytes into data queue */
            dat_vect(k+off)=receive_data_block.data(k+start);
        end;
    end;
end;

    /* update local evc value */
    do while (local_evc_value < remote_evc_value);

        call advance (evcid);
        local_evc_value = add2bit16(local_evc_value,'0001'b4);
    end;
end;

    else do; /* if not evc */
    end;

    call disable_cpu_interrupts;

end;

end process_packet;

```

APPENDIX D

THE DRIVER

The driver is the software link between a cluster and the ECCB that hooks up a cluster onto the Ethernet. It manages all cluster external message exchange (transmission and reception), and sets up the clusters communication ability in the first place.

The driver resides in the user area of SBC 1, which is dedicated to serve as the cluster's host. The executable file names are C1PROC.COMD and C2PROC.COMD for the two cluster constellation of RTC* in the AEGIS lab at the US Naval Postgraduate School.

The LINK86 Input option is used to link files sysinit1, sysdev, asmrout, and gatedmod into C1PROC. Sysinit2, sysdev, asmrout, and gatedmod are linked into C2PROC.

Cluster specific information about the creation and distribution of eventcounts is contributed by sysinit1 and sysinit2 respectively, which are the main procedures for the linkage. Additional cluster specific information is read from the address.dat file and the relation.dat file during execution at runtime. The %included files sysdef.pli and NI3010.dcl provide system-wide information.

```

*****
*****
***          C1PROC.INP file          ***
***-----
*** This file is used to link system initialization, ***
*** driver, assembly language routines, and gatemod ***
*** into C1PROC.COM, the executable file run on SBC 1 at ***
*** cluster 1. ***
*****

```

```

c1proc =
sysinit1 [code[ab[439]],data[ab[800],m[0],ad[82]],map[all]],
sysdev,
asmrout,
gatemod

```

```

*****
*****
***          Cluster 1 RELATION.DAT file          ***
***-----
*** This file keeps the data used to build the relation ***
*** table: ***
*** ***
***   evc_id      numdat      point      alen      bytes      ***
*** ***
***   col         col         col         col         col         ***
***   5           15          25          35          45          ***
*****

```

01	1	8c58	50	6
03	1	8d84	50	9
00	0	0000	00	0

```

*****
*****
***          Cluster 1 ADDRESS.DAT file          ***
***-----
*** This file keeps the number of group addresses, ***
*** the cluster's group address(es) and the cluster's ***
*** source address. ***
*****

```

```

1,
'00000000'b,'00000001'b,
'00000000'b,'00000001'b

```

```

*****

```

```

*****
*****
***      SYSINIT1.PLI file      ***
***-----***
*** This is the system initialization procedure for ***
*** cluster 1. ***
*****

```

```

sysinit1: proc options (main);

    %include 'sysdef.pli';
    %replace

        EVC_TYPE      by '00'b4;

    /* main */

    call define_cluster ('0001'b4);
        /* must be called prior to creating evc's */

    /**** USER ****/

    call create_evc (TRACK_IN);
    call create_evc (TRACK_OUT);
    call create_evc (MISSILE_ORDER_IN);
    call create_evc (MISSILE_ORDER_OUT);

    /*** SYSTEM ***/

    call create_evc (ERB_READ);
    call create_evc (ERB_WRITE);
    call create_seq (ERB_WRITE_REQUEST);

    /* distrib. map called after eventcounts have
       been created */

    call distribution_map (EVC_TYPE, TRACK_IN, '0003'b4);
        /* local and remote copy of TRACK_IN needed */
    call distribution_map (EVC_TYPE, MISSILE_ORDER_OUT,
        '0003'b4);
        /* local and remote copy of MISSILE_ORDER_OUT
           needed */

        /* create driver */
    call create_proc ('fc'b4, '80'b4,
        '26a5'b4, '0800'b4, '205f'b4,
        '0439'b4, '0800'b4, '2800'b4);
    call await ('fe'b4, '01'b4);

end sysinit1;
*****

```

```

*****
*****
***          C2PROC.INP file          ***
***-----
*** This file is used to link system initializatio, ***
*** driver, assembly language routines, and gatmodule ***
*** into C2PROC.CMD, the executable file run on SBC 1 at ***
*** cluster 2. ***
*****

```

```

c2proc =
sysinit2 [code[ab[439]],data[ab[800],m[0],ad[82]],map[all]],
sysdev,
asmrout,
gatmod

```

```

*****
*****
***          Cluster 2 RELATION.DAT file          ***
***-----
*** This file keeps the data used to build the relation ***
*** table: ***
***
***   evc_id      numdat      point      olen      bytes      ***
***
***   col         col         col         col         col         ***
***   5           15          25          35          45          ***
*****

```

01	1	8c58	50	6
05	1	8d94	20	3
03	1	8dcd	50	9
00	0	0000	00	0

```

*****
*****
***          Cluster 2 ADDRESS.DAT file          ***
***-----
*** This file keeps the number of group addresses, ***
*** the cluster's group address(es),and the cluster's ***
*** source address. ***
*****

```

```

1.
'00000000'b,'00000010'b,
'00000000'b,'00000010'b

```

```

*****

```

```

*****
*****
***      SYSINIT2.PLI file      ***
***-----***
*** This is the system initialization procedure for ***
*** cluster 2. ***
*****

```

```

sysinit2: proc options (main);

%include 'sysdef.pli';

%replace

    EVC_TYPE          by '00'b4;

/* main */

    call define_cluster ('0002'b4); /* must be called prior
                                     to creating evc's */

    /**** USER ****/

    call create_evc (TRACK_IN);
    call create_evc (TRACK_OUT);
    call create_evc (MISSILE_ORDER_IN);
    call create_evc (MISSILE_ORDER_OUT);
    call create_evc (DELTA_IN);
    call create_evc (DELTA_OUT);

    /*** SYSTEM ***/

    call create_evc (ERB_READ);
    call create_evc (ERB_WRITE);
    call create_seq (ERB_WRITE_REQUEST);

        /* distrib. map called after eventcounts have
           been created */

    call distribution_map (EVC_TYPE, TRACK_OUT, '0003'b4);
        /* local and remote copy of TRACK_IN needed */
    call distribution_map (EVC_TYPE, MISSILE_ORDER_IN,
        '0003'b4);
        /* local and remote copy of MISSILE_ORDER_IN needed */
    call create_proc ('fc'b4, '80'b4,
        '26a5'b4, '0800'b4, '006b'b4,
        '0439'b4, '0800'b4, '0800'b4);

    call await ('fe'b4, '01'b4);

end sysinit2;
*****

```

```

/*****
/*****
** SYSDEF   FILE SYSDEF.PLI   David J. BREWER  1 SEP 84 **
**=====
** This section of code is given as a PLI file to be
** %INCLUDE'd with MCORTEX user programs.  ENTRY
** declarations are made for all available MCORTEX
** functions.
**
*****/
/*****

```

DECLARE

```

    advance ENTRY (BIT (8)),
        /* advance (event_count_id) */

    await ENTRY (BIT (8), BIT (16)),
        /* await (event_count_id, awaited_value) */

    create_evc ENTRY (BIT (8)),
        /* create_evc (event_count_id) */

    create_proc ENTRY (BIT (8), BIT (8),
                        BIT (16), BIT (16), BIT (16),
                        BIT (16), BIT (16), BIT (16)),
        /* create_proc (processor_id, processor_priority,
        /* stack_pointer_highest, stack_seg, ip
        /* code_seg, data_seg, extra_seg) */

    create_seq ENTRY (BIT (8)),
        /* create_seq (sequence_id) */

    preempt ENTRY (BIT (8)),
        /* preempt (processor_id) */

    read ENTRY (BIT (8)) RETURNS (BIT (16)),
        /* read (event_count_id) */
        /* RETURNS current_event_count */

    ticket ENTRY (BIT (8)) RETURNS (BIT (16)),
        /* ticket (sequence_id) */
        /* RETURNS unique_ticket_value */

    define_cluster ENTRY (bit (16)),
        /* define_cluster (local_cluster_address) */

    distribution_map ENTRY (bit (8), bit (8), bit (16)),
        /* distribution_map (distribution_type, id,
        cluster_addr) */

```

```

add2bit16 ENTRY (BIT(16), BIT(16)) RETURNS (BIT (16));
/* add2bit16 ( a_16bit_#, another_16bit_#) */
/* RETURNS a_16bit_# + another_16bit_# */

```

```
%replace
```

```

/*-----
*** EVCSID's ***

```

```

(1) USER */

```

```

TRACK_IN          by '01'b4,
TRACK_OUT         by '02'b4,
MISSILE_ORDER_IN  by '03'b4,
MISSILE_ORDER_OUT by '04'b4,
DELTA_IN          by '05'b4,
DELTA_OUT         by '06'b4,

```

```

/* (2) SYSTEM */

```

```

ERB_READ          by 'fc'b4,
ERB_WRITE         by 'fd'b4,

```

```

/*-----
*** SEQUENCER NAMES ***

```

```

(1) USER */

```

```

/* (2) SYSTEM */

```

```

ERB_WRITE_REQUEST by 'ff'b4,

```

```

/*-----

```

```

*** SHARED VARIABLE POINTERS ***

```

```

(1) USER */

```

```

/* (2) SYSTEM */

```

```

block_ptr_value  by '8000'b4,
xmit_ptr_value   by '8078'b4,
rcv_ptr_value    by '8666'b4,

```

```

END_RESERVE      by 'FFFF'b4;

```

```

*****

```

```

*****
*****
***      NI3010.DCL file      ***
*****

```

```
%replace
```

```
/*      I/O port addresses
```

These values are specific to the use of the INTERLAN
NI3010 MULTIBUS to ETHERNET interface board. Any change
to the I/O port address of '00b0' hex (done so with a DIP
switch) will require a change to these addresses to
reflect that change.

```
*/
```

```

command_register      by 'b0'b4,
command_status_register  by 'b1'b4,
transmit_data_register  by 'b2'b4,
interrupt_status_reg   by 'b5'b4,
interrupt_enable_register by 'b8'b4,
high_byte_count_reg    by 'bc'b4,
low_byte_count_reg     by 'bd'b4,

```

```
/* end of I/O port addresses */
```

```
/* Interrupt enable status register values */
```

```

disable_ni3010_interrupts  by '00'b4,
ni3010_intrpts_disabled    by '00'b4,
receive_block_available     by '04'b4,
transmit_dma_done           by '06'b4,
receive_dma_done            by '07'b4,

```

```
/* end register values */
```

```
/* Command Function Codes */
```

```

module_interface_loopback  by '01'b4,
internal_loopback          by '02'b4,
clear_loopback             by '03'b4,
go_offline                 by '08'b4,
go_online                  by '09'b4,
onboard_diagnostic         by '0a'b4,
clr_insert_source          by '0e'b4,
load_transmit_data         by '28'b4,
load_and_send              by '29'b4,
load_group_addresses       by '2a'b4,
reset                     by '3f'b4;

```

```
/* end Command Function Codes */
```

```
*****
```



```

*****
*****
***      SYSDEV.PLI file      (driver)      ***
*****

```

```
sysdev: procedure;
```

```
/* Date:          25 NOVEMBER 1985
```

```
Programmer:      Reinhard HAEGER
```

```
Module Function: To serve as the Ethernet Communication
                  Controller Board (NI3010) device
                  handler (driver). This process is
                  scheduled under MCORTEX and consumes
                  Ethernet Requests Packets (ERP)
                  generated by the SYSTEM$IO routine in
                  LEVEL2.SRC.
                  It creates a relation table that keeps
                  information about the interrelationship
                  between eventcounts and user shared
                  data, and uses this information for
                  producing and processing Ethernet
                  messages.
```

This driver is the version provided by David Brewer modified in order to ensure system-wide data sharing. The Transmit_Data_Block and the Receive_Data_Block were changed to keep 1500 bytes of data. New procedures Make_Table and Make_Message were added, and procedure Process_Packet was completely changed to provide cluster external user shared data exchange. Procedure Transmit_Packet was modified to provide flexible exact message length to the ECCB if the length is greater than 60 bytes, and minimum message length if the message is 60 bytes or less.

```
*/
```

```
%replace
```

```

    evc_type          by '00'b4,
    erb_block_len     by 20,
    erb_block_len_m1  by 19,
    infinity          by 32767;

```

```
%include 'sysdef.pli';
```

DECLARE

```

1 erb(0:erb_block_len_m1) based (block_ptr),
  2 command      bit (8),
  2 type_name    bit (8),
  2 name_value   bit (16),
  2 remote_addr  bit (16),

```

```

1 transmit_data_block based (xmit_ptr),
  2 destination_address_a  bit (8),
  2 destination_address_b  bit (8),
  2 destination_address_c  bit (8),
  2 destination_address_d  bit (8),
  2 destination_address_e  bit (8),
  2 destination_address_f  bit (8),
  2 source_address_a       bit (8),
  2 source_address_b       bit (8),
  2 source_address_c       bit (8),
  2 source_address_d       bit (8),
  2 source_address_e       bit (8),
  2 source_address_f       bit (8),
  2 type_field_a           bit (8),
  2 type_field_b           bit (8),
  2 data (1500)            bit (8),

```

```

1 receive_data_block based (rcv_ptr),

```

```

  2 frame_status           bit (8),
  2 null_byte              bit (8),
  2 frame_length_lsb       bit (8),
  2 frame_length_msb       bit (8),
  2 destination_address_a  bit (8),
  2 destination_address_b  bit (8),
  2 destination_address_c  bit (8),
  2 destination_address_d  bit (8),
  2 destination_address_e  bit (8),
  2 destination_address_f  bit (8),
  2 source_address_a       bit (8),
  2 source_address_b       bit (8),
  2 source_address_c       bit (8),
  2 source_address_d       bit (8),
  2 source_address_e       bit (8),
  2 source_address_f       bit (8),
  2 type_field_a           bit (8),
  2 type_field_b           bit (8),
  2 data(1500)             bit (8),
  2 crc_msb                bit (8),
  2 crc_upper_middle_byte  bit (8),
  2 crc_lower_middle_byte  bit (8),
  2 crc_lsb                bit (8),

```

```

    e_ptr pointer,
    erp_vect(4) bit (8) based (e_ptr),
    (maxqbytes,bytecount) fixed bin (15).

1 rel_tab(100),
  2 evc_id bit (8),
  2 numdat fixed bin (7),
  2 data(10),
    3 point pointer,
    3 qlen fixed bin (7),
    3 bytes fixed bin (15),
    3 next_out fixed bin (7),
    3 next_in fixed bin (7),

    (xmit_ptr,rcv_ptr,block_ptr) pointer,
    index fixed bin (15),
    (addr_e,addr_f) bit (8),
    address file,
    copy_ie_register bit (8),
    (cluster_addr,erb_write_value,i) bit (16),
    (j,k) fixed bin (15),
    reg_value bit (8),
    write_io_port entry (bit (8), bit (8)),
    read_io_port entry (bit (8), bit (8)),
    initialize_cpu_interrupts entry,
    enable_cpu_interrupts entry,
    disable_cpu_interrupts entry,
    write_bar entry (bit(16));

/* end declaration */

%replace

/* codes specific to the Intel 8259a Programmable
Interrupt Controller (PIC) */

/* note that */ icw1_port_address by 'c0'b4.
/* icw2,icw4,*/ icw2_port_address by 'c2'b4,
/* and ocw */ icw4_port_address by 'c2'b4,
/* use same */ ocw_port_address by 'c2'b4,
/* port addr */

/* note: icw ==> initialization
control
word

ocw ==> operational
command
word */

```

```

        icw1                                by '13'b4,
/* single PIC configuration, edge
   triggered input */

        icw2                                by '40'b4,
/* most significant bits of vectoring
   byte; for an interrupt 5,
   the effective address will be
   (icw2 + interrupt #) * 4 which
   will be (40 hex + 5) * 4 = 114 hex
   */

        icw4                                by '0f'b4,
/* automatic end of interrupt
   and buffered mode/master */

        ocw1                                by '8f'b4;

/* unmask interrupt 4 (bit 4), */
/* interrupt 5 (bit 5), and */
/* interrupt 6 (bit 6), mask all others */

        /* end 8259a codes */

/* include constants specific to the NI3010
   board */

#include 'ni3010.dcl';

/*****

/* Main Body */

/* check message
   put skip(2) list('starting make_table'); */

call make_table;

/* check message
   put skip(2) list('make_table done'); */

call write_io_port(interrupt_enable_register,
                   disable_ni3010_interrupts);
call initialize_pic;
call initialize_cpu_interrupts;
call read_io_port (command_status_register.reg_value);
call perform_command (reset);

```

```

call program_group_addresses;
/* assignments to the source and destination address
   fields that will not change */

call perform_command (clr_insert_source);
/* NI3010 performance is enhanced in this mode */

unspec(block_ptr) = block_ptr_value;
unspec(rcv_ptr) = rcv_ptr_value;
unspec(xmit_ptr) = xmit_ptr_value;

/* make one time assignments to transmit data block */

transmit_data_block.destination_address_a = '03'b4;
transmit_data_block.destination_address_b = '00'b4;
transmit_data_block.destination_address_c = '00'b4;
transmit_data_block.destination_address_d = '00'b4;
transmit_data_block.source_address_a = '03'b4;
transmit_data_block.source_address_b = '00'b4;
transmit_data_block.source_address_c = '00'b4;
transmit_data_block.source_address_d = '00'b4;

/* get the local cluster address - file was
   opened in proc program_group_addresses */

get file (address) list (addr_e, addr_f);
transmit_data_block.source_address_e = addr_e;
transmit_data_block.source_address_f = addr_f;

cluster_addr = addr_e || addr_f;
put skip (2) edit ('*** CLUSTER ',cluster_addr,
                  ' Initialization Complete ***')
              (col(15),a,b4(4),a);

i = '0001'b4;
call perform_command (go_online);

/* at this point copy_ie_reg = RBA . but
   ie_reg on NI3010 is actually disabled */
call disable_cpu_interrupts;

do k = 1 to infinity;
/* note: interrupt not allowed during a
   call to MCCORTEX primitive */

erb_write_value = read(ERB_WRITE);
/* In the MXTRACE version of the RTOS
   all primitive calls clear and
   set interrupts (diagnostic message
   routines), so the NI3010 interrupts
   must be disabled on entry to MXTRACE */

```

```

do while (erb_write_value < 1);
/* busy waiting */
erb_write_value = read(ERB_WRITE);
copy_ie_register=receive_block_available;
call write_io_port(interrupt_enable_register,
                    receive_block_available);
call enable_cpu_interrupts;
/* if a packet has been received, this
   is when an interrupt may occur - can
   see that outbound packets are always
   favored. */
do j = 1 to 1000;
/* interrupt window for packets received */
end; /* do j */
call disable_cpu_interrupts;
if (copy_ie_register = receive_dma_done) then
do;
/* receive DMA operation started, so let
   finish. */
call enable_cpu_interrupts;
do while (copy_ie_register = receive_dma_done);
end;
call disable_cpu_interrupts;
end; /* ift */
copy_ie_register = disable_ni3010_interrupts;
call write_io_port(interrupt_enable_register,
                    disable_ni3010_interrupts);

end; /* busy */

/* ERB has an ERP in it, so process it */
/* no external interrupts (RBA) until
   the ERP is consumed and the packet
   gets sent */
index = mod((fixed(1) - 1), erb_block_len);
/* 32k limit */

transmit_data_block.destination_address_e=
    substr(erb(index).remote_addr, 1,8);
transmit_data_block.destination_address_f=
    substr(erb(index).remote_addr, 9,8);

/* put overlay over ERP */
e_ptr=addr(erb(index).command);

call make_message;
call advance (ERB_READ);
/* caution here !!!!
   an ADVANCE will result in a call to VP$SCHEDULER,
   which will set CPU interrupts on exit.
   It's the reason NI3010 interrupts are disabled
   first in the Do While loop above. */

```

```

        /* packet ready to go, so send it */
        call transmit_packet (bytecount);

        /* copy_ie_register=RBA, but not actual register */
        call disable_cpu_interrupts;

        /* setting up for next ERP consumption */
        i = add2bit16(i, '0001'b4);

    end; /* do forever */
        /* end main body */

/*****

make_table: procedure ;

declare
    relation file,
    (j,i) fixed bin (15),
    eof bit(8);

open file (relation) stream input;
i=0;
eof='0'b4;
do while (eof='0'b4) ;
    i=i+1;
    /* read data from relation.dat file */
    get file (relation) edit(rel_tab(i).evc_id,
                            rel_tab(i).numdat)
                        (column(5),b4(2),column(15),f(2));
    do j=1 to (rel_tab(i).numdat);
        /* read data for all related items */
        get file (relation) edit
                        (unspec(rel_tab(i).data(j).point),
                         rel_tab(i).data(j).qlen,
                         rel_tab(i).data(j).bytes)
                        (column(25),b4(4),column(35),f(4),column(45),f(4));
        maxbytes=max(maxbytes,
                     rel_tab(i).data(j).bytes*rel_tab(i).data(j).qlen);
    end;
    /* if sentinel is reached */
    if rel_tab(i).evc_id = '00'b4 then
    do;
        eof='1'b4;
        put skip list('longest data queue at this cluster:',
                     maxbytes,' bytes');
    end;
end;
end make_table;

*****/

```

```

make_message: procedure ;

declare
    dat_ptr pointer,
    dat_vect(1500) bit (8) based (dat_ptr),
    (next_out,r,j) fixed bin (7),
    (off,start,last,k) fixed bin (15);

    /* check for evc */
if erp_vect(1)=EVC_TYPE then
do;
    bytecount=4;
    do k=1 to 4;
        /* put evc info into data field */
        transmit_data_block.data(k)=erp_vect(k);
    end;

    /* check message
put skip list('::::',transmit_data_block.data(1),'::',
               transmit_data_block.data(2),'::',
               transmit_data_block.data(3),'::',
               transmit_data_block.data(4),':::::');
    */

    r=1;
    /* find respective relation table entry */
    do while ((rel_tab(r).evc_id ^= erp_vect(2)) &
              (rel_tab(r).evc_id ^= '00'b4));
        r=r+1;
    end;

    /* if evc entry */
if rel_tab(r).evc_id ^= '00'b4 then
do;
    /* for every related item */
    do j=1 to rel_tab(r).numdat;
        start=bytecount + 1; /* find where to put */
        last=rel_tab(r).data(j).bytes; /* how long it is */
        bytecount=bytecount + last; /* keep track */
        dat_ptr=rel_tab(r).data(j).point; /*align datvect*/
        next_out=rel_tab(r).data(j).next_out;
        /* compute offset of item in data queue */
        off=(rel_tab(r).data(j).bytes * next_out) + 1;
        /* compute next slot number to go */
        rel_tab(r).data(j).next_out=mod(next_out + 1,
                                         rel_tab(r).data(j).qlen);
        do k=0 to (last-1);
            /* put item's bytes into data field */
            transmit_data_block.data(k+start)=dat_vect(k+off);
        end;
    end;
end;
end;

```



```

        /* compute total bytecount for message */
        bytecount=bytecount + 14;
    end;
    else do;    /* if not evc */
    end;
    end make_message;

/*****

initialize_pic:    procedure;

DECLARE
    write_io_port entry (bit (8) , bit(8));
    call write_io_port (icw1_port_address,icw1);
    call write_io_port (icw2_port_address,icw2);
    call write_io_port (icw4_port_address,icw4);
    call write_io_port (ocw_port_address,ocw1);

    end initialize_pic;

/*****

perform_command:    procedure (command);

DECLARE
    command bit (8) ,
    reg_value bit (8) ,
    srf bit (8) ,
    write_io_port entry (bit (8) , bit (8) ),
    read_io_port  entry (bit (8) , bit (8) );

    srf = '0'b4;
    call write_io_port (command_register,command);
    do while ((srf & '01'b4) = '00'b4);
        call read_io_port (interrupt_status_reg, srf);
    end; /* do while */
    call read_io_port (command_status_register, reg_value);
    if (reg_value > '01'b4) then
    do;
        /* not (SUCCESS or SUCCESS with Retries) */
        put skip edit ('*** ETHERNET Board Failure ***')
            (col(20),a);
        /* when this occurs, run the diagnostic
           routine T3010/Cx, where x is the
           current cluster number */

        stop;
    end; /* itd */

    end perform_command;

/*****

```

```
transmit_packet: procedure (byte_count) external;
```

```
DECLARE
```

```
  p pointer,  
  byte_count fixed bin (15) ,  
  bytevector(2) bit (8) based (p),  
  srf bit (8) ,  
  reg_value bit (8) ,  
  write_io_port entry (bit (8), bit (8) ),  
  read_io_port entry (bit (8), bit (8) ),  
  enable_cpu_interrupts      entry,  
  disable_cpu_interrupts     entry,  
  write_bar entry (bit(16));
```

```
  /* begin */
```

```
srf = '0'b4;
```

```
call write_bar (xmit_ptr_value);
```

```
  /* if message longer than minimum size */
```

```
if (byte_count > 60) then
```

```
do;
```

```
  p=addr(byte_count);
```

```
  /* call with exact bytecount */
```

```
  call write_io_port(high_byte_count_reg,bytevector(2));
```

```
  call write_io_port(low_byte_count_reg,bytevector(1));
```

```
end;
```

```
  /* if message is not longer than minimum size */  
else do;
```

```
  /* call with minimum bytecount of 60 */
```

```
  call write_io_port(high_byte_count_reg,'30'b4);
```

```
  call write_io_port(low_byte_count_reg,'3c'b4);
```

```
end;
```

```
copy_ie_register = transmit_dma_done;
```

```
call write_io_port(interrupt_enable_register,  
  transmit_dma_done);
```

```
call enable_cpu_interrupts;
```

```
do while (copy_ie_register = transmit_dma_done);
```

```
end;  /* loop until the interrupt handler  
      takes care of the TDD interrupt -  
      it sets copy_ie_register = RBA */
```

```
call perform_command (load_and_send);
```

```
put skip list('transmitting');
```

```
end transmit_packet;
```

```
/******
```

```

HL_interrupt_handler: procedure external;

/* This routine is called from the low level
   8086 assembly language interrupt routine */

DECLARE

    write_io_port entry (bit (8), bit (8) ),
    read_io_port entry (bit (8) , bit (8) ),
    enable_cpu_interrupts entry,
    disable_cpu_interrupts entry,
    write_bar entry (bit(16));

/* begin */

call write_io_port(interrupt_enable_register,
                   disable_ni3010_interrupts);

if (copy_ie_register = receive_block_available) then
do;

    call write_bar (rcv_ptr_value);
    call write_io_port(high_byte_count_reg,'25'b4);
    call write_io_port(low_byte_count_reg,'f2'b4);

    /* initiate receive DMA */

    copy_ie_register = receive_dma_done;
    call write_io_port(interrupt_enable_register,
                       receive_dma_done);

end; /* do */
else
if (copy_ie_register = receive_dma_done) then
do;
    call process_packet;
    copy_ie_register = receive_block_available;
    call write_io_port(interrupt_enable_register,
                       receive_block_available);
end; /* if then do */
else
if (copy_ie_register = transmit_dma_done) then
do;

    copy_ie_register = receive_block_available;
    /* NI3010 interrupts disabled on entry */
end; /* if then do */

end HL_interrupt_handler;

/*****/

```

```

process_packet: procedure;

DECLARE

    evcid bit (8),
    local_evc_value bit (16),
    (dat_ptr,p) pointer,
    remote_evc_value bit (16) based (p),
    dat_vect(1500) bit (8) based (dat_ptr),
    (off,start,last,k) fixed bin (15),
    (next_in,r,j) fixed bin (7);
put skip list('receiving');
/* check for evc */
if receive_data_block.data(1)=evc_type then
do;
    p = addr(receive_data_block.data(3));
    evcid=receive_data_block.data(2);
    local_evc_value = read(evcid);
    if local_evc_value < remote_evc_value then
do;
        r=1;

        /* find evc entry in relation table */
do while ((rel_tab(r).evc_id ^= evcid) &
            (rel_tab(r).evc_id ^= '00'b4));
            r=r+1;
end;
if rel_tab(r).evc_id ^= '00'b4 then
do;
        bytecount=4; /* jump over evc info */

do j=1 to rel_tab(r).numdat;
        start=bytecount+1; /* compute start of item */
        last=rel_tab(r).data(j).bytes; /* and length */
        bytecount=bytecount+last; /* and item's end */
        next_in=rel_tab(r).data(j).next_in;
        /* compute offset in data queue */
        off=(last*next_in) + 1;
        /* compute next slot number to fill */
        rel_tab(r).data(j).next_in=mod(next_in+1,
            rel_tab(r).data(j).qlen);

        dat_ptr=rel_tab(r).data(j).point; /* align datvect*/

do k=0 to (last-1);

        /* put item bytes into data queue */
        dat_vect(k+off)=receive_data_block.data(k+start);
end;
end;
end;
end;

```

```

    /* update local evc value */
    do while (local_evc_value < remote_evc_value);

        call advance (evcid);
        local_evc_value = add2bit16(local_evc_value,'0221'b4);
    end;
end;

else do;    /* if not evc */
end;

call disable_cpu_interrupts;

end;

end process_packet;

/*****

program_group_addresses: procedure;

DECLARE

    1 group_addr(40) based (group_ptr),
      2 mc_group_field_a    bit (8),
      2 mc_group_field_b    bit (8),
      2 mc_group_field_c    bit (8),
      2 mc_group_field_d    bit (8),
      2 mc_group_field_e    bit (8),
      2 mc_group_field_f    bit (8);

DECLARE

    (group_ptr,p) pointer.
    (field_e, field_f) bit (8),
    bit_8_groups bit (8) based (p),
    (i.num_groups.groups_times_6) fixed bin (7);

unspec(group_ptr) = xmit_ptr_value;
open file (address) stream input;
get file (address) list (num_groups);
do i = 1 to num_groups;

    group_addr(i).mc_group_field_a = '03'b4;
    group_addr(i).mc_group_field_b = '00'b4;
    group_addr(i).mc_group_field_c = '00'b4;
    group_addr(i).mc_group_field_d = '00'b4;

```

```

    get file (address) list (field_e,field_f);
    group_addr(i).mc_group_field_e = field_e;
    group_addr(i).mc_group_field_f = field_f;

end;      /* do i */

call disable_cpu_interrupts;
call write_bar (xmit_ptr_value);
call write_io_port(high_byte_count_reg, '00'b4);
groups_times_6 = 6 * num_groups;
p = addr (groups_times_6);
call write_io_port(low_byte_count_reg, bit_8_groups);
copy_ie_register = transmit_dma_done;
call write_io_port(interrupt_enable_register,
    transmit_dma_done);
call enable_cpu_interrupts;
do while (copy_ie_register = transmit_dma_done);
end;      /* loop until the interrupt handler
           takes care of the TDD interrupt -
           it sets COPY_IE_REG = RBA */

call perform_command(load_group_addresses);

end program_group_addresses;

/*****/

end;      /* system device handler and packet processor
           (driver) */

*****/
*****
*****
***      ASMROUT.A86 file      ***
*****

extrn hl_interrupt_handler : far

public write_io_port
public read_io_port
public write_bar
public initialize_cpu_interrupts
public enable_cpu_interrupts
public disable_cpu_interrupts
;*****

```

write_io_port:

; Parameter Passing Specification:

```
;
;          entry          exit
;
; parameter 1    <port address>    <unchanged>
;
; parameter 2    <value to be outputted> <unchanged>
;
```

```
dseg
port_address    rb    1

cseg
push bx! push si! push dx! push ax
mov  si, [bx]
mov  al, [si]
mov  port_address, al
mov  si, 2[bx]
mov  al, [si]
mov  dl, port_address
mov  dh, 00h
out  dx, al
pop  ax! pop dx! pop si! pop bx
ret
```

read_io_port:

; Parameter Passing Specification

```
;
;          entry          exit
;
; parameter 1    <port address>    <unchanged>
; parameter 2    <meaningless>     <register value>
;
```

```
cseg
push bx! push si! push dx! push ax
mov  si, [bx]
mov  al, [si]
mov  port_address, al
mov  si, 2[bx]
mov  dl, port_address
mov  dh, 00h
in   al, dx
mov  [si], al
pop  ax! pop dx! pop si! pop bx!
ret
```

write_bar:

; Parameter Passing Specification

; parameter 1 (and only): the address of the data block
; to be transmitted or received.

dseg

e_bar_port equ 0b9h
h_bar_port equ 0bah
l_bar_port equ 0bbh
temp_e_byte rb 1
temp_es rw 1

cseg

; This module computes a 24 bit address from a 32 bit
; address - actually it's a combination of the ES
; register and the IP passed via a parameter list.

push bx! push ax! push cx! push es! push dx! push si

mov dx, 0000h ; shared memory segment
mov es, dx
mov temp_es, es
mov dx, es
mov si, [bx]
mov ax, [si]
mov cl, 12
shr dx, cl
mov temp_e_byte, dl
mov ix, temp_es
mov cl, 4
shl dx, cl
add ax, dx
jnc no_add
add_1: inc temp_e_byte
no_add: out l_bar_port, al
mov al, ah
out h_bar_port, al
mov al, temp_e_byte
out e_bar_port, al
pop si! pop dx! pop es! pop cx! pop ax! pop bx
ret

;*****

initialize_cpu_interrupts:

; Module Interface Specification:

; Caller: Ethertest(PL/I) Procedure

; Parameters: NONE

initmodule cseg common

org 114h

int5_offset rw 1

int5_segment rw 1

cseg

push bx

push ax

mov bx, offset interrupt_handler

mov ax, 0

push ds

mov ds, ax

mov ds:int5_offset, bx

mov bx, cs

mov ds:int5_segment, bx

pop ds

pop ax

pop bx

sti

ret

;*****

enable_cpu_interrupts:

; Module Interface Specification:

; Caller: Ethertest(PL/I) Procedure

; Parameters: NONE

sti

ret

;*****

disable_cpu_interrupts:

; Module Interface Specification:

; Caller: Ethertest(PL/I) Procedure

; Parameters: none

cli
ret

interrupt_handler:

; IP, CS, and flags are already on stack
; save all other registers

push ax
push bx
push cx
push dx
push si
push di
push bp
push ds
push es
call hl_interrupt_handler
; high level source routine
; In Ethertest Module (PL/I)
; restore registers

pop es
pop ds
pop bp
pop di
pop si
pop dx
pop cx
pop bx
pop ax
sti
iret

end

```

;*****
;*****/
;* GATEMOD / GATETRC   File GATEM/T.a86   BREWER 1 SEP 84 */
;-----*/
;* This module is given to the user in obj form to link */
;* with his initial and process modules. Any changes to */
;* user services available from the OS must be reflected */
;* here. In this way the user need not be concerned with */
;* actual GATEKEEPER services codes. Two lines of code */
;* are contained in conditional assembly statements and */
;* control the output to be GATEMOD or GATETRC depending */
;* on the value of GATEMOD at the code start. */
;-----*/
;* This module reconciles parameter passing anomalies */
;* between MCORTEX (written in PL/M) and user programs */
;* (written in PL/I). */
;-----*/
;* All calls are made to the GATEKEEPER in LEVEL2 of the */
;* OS. The address of the GATEKEEPER must be given below.*/
;-----*/
;* The ADD2BIT16 function does not make calls to MCORTEX. */
;* It's purpose is to allow the addition of two unsigned */
;* 16 bit numbers from PL/I programs. */
;*****/

```

DSEG

```

GATEMOD EQU 0 ;*** SET TO ZERO FOR GATETRC
               ;*** SET TO ONE FOR GATEMOD
PUBLIC ADVANCE ;*** THESE DECLARATIONS MAKE THE
PUBLIC AWAIT   ;*** GATEKEEPER FUNCTIONS VISIBLE
PUBLIC CREATE_EVC ;*** TO EXTERNAL PROCESSES
PUBLIC CREATE_PROC
PUBLIC CREATE_SEQ
PUBLIC PREEMPT
PUBLIC READ
PUBLIC TICKET
PUBLIC DEFINE_CLUSTER
PUBLIC DISTRIBUTION_MAP
PUBLIC ADD2BIT16

AWAIT_IND EQU 0 ;*** THESE ARE THE IDENTIFICATION
ADVANCE_IND EQU 1 ;*** CODES RECOGNIZED BY THE
CREATE_EVC_IND EQU 2 ;*** GATEKEEPER IN LEVEL II OF
CREATE_SEQ_IND EQU 3 ;*** MCORTEX
TICKET_IND EQU 4
READ_IND EQU 5
CREATE_PROC_IND EQU 6
PREEMPT_IND EQU 7
DEFINE_CLUSTER_IND EQU 8
DISTRIBUTION_MAP_IND EQU 9

```

```

IF GATEMOD
GATEKEEPER_IP DW 0036H
GATEKEEPER_CS DW 0BADH
ELSE
GATEKEEPER_IP DW 0068H          ;#### 1 #### <-----
GATEKEEPER_CS DW 0B4CH          ;#### 2 #### <-----
ENDIF
GATEKEEPER EQU DWORD PTR GATEKEEPER_IP

```

CSEG

;*** AWAIT *** AWAIT *** AWAIT *** AWAIT *** AWAIT *****/

AWAIT:

```

PUSH ES
MOV SI,2[BX]          ;SI <-- PNT TO COUNT AWAITED
MOV BX,[BX]           ;BX <-- PNT TO NAME OF EVENT
MOV AL,AWAIT_IND
PUSH AX               ;N <-- AWAIT INDICATOR
MOV AL,[BX]
PUSH AX               ;BYT <-- NAME OF EVENT
MOV AX,[SI]           ;AX <-- COUNT AWAITED
PUSH AX               ;WORDS <-- COUNT AWAITED
PUSH AX               ;PTR_SEG <-- UNUSED WORD
PUSH AX               ;PTR_OFFSET <--UNUSED WORD
CALLF GATEKEEPER
POP ES

```

RET

;*** ADVANCE *** ADVANCE *** ADVANCE *** ADVANCE *****/

ADVANCE:

```

PUSH ES
MOV BX,[BX]           ;BX <-- PTR TO NAME OF EVENT
MOV AL,ADVANCE_IND
PUSH AX               ;N <-- ADVANCE INDICATER
MOV AL,[BX]
PUSH AX               ;BYT <-- NAME OF EVENT
PUSH AX               ;WORDS <-- UNUSED WORD
PUSH AX               ;PTR_SEG <-- UNUSED WORD
PUSH AX               ;PTR_OFFSET <--UNUSED WORD
CALLF GATEKEEPER
POP ES

```

RET

;*** CREATE_EVC *** CREATE_EVC *** CREATE_EVC *****/

CREATE_EVC:

PUSH ES	
MOV BX,[BX]	;BX <-- PTR TO NAME OF EVENT
MOV AL,CREATE_EVC_IND	
PUSH AX	;N <-- CREATE_EVC INDICATOR
MOV AL,[BX]	
PUSH AX	;BYT <-- NAME OF EVENT
PUSH AX	;WORDS <-- UNUSED WORD
PUSH AX	;PTR_SEG <-- UNUSED WORD
PUSH AX	;PTR_OFFSET <--UNUSED WORD
CALLF GATEKEEPER	
POP ES	

RET

;*** CREATE_SEQ *** CREATE_SEQ *** CREATE_SEQ *****/

CREATE_SEQ:

PUSH ES	
MOV BX,[BX]	;BX <-- PTR TO NAME OF SEQ
MOV AL,CREATE_SEQ_IND	
PUSH AX	;N <-- CREATE_SEQ INDICATER
MOV AL,[BX]	
PUSH AX	;BYT <-- NAME OF SEQ
PUSH AX	;WORDS <-- UNUSED WORD
PUSH AX	;PTR_SEG <-- UNUSED WORD
PUSH AX	;PTR_OFFSET <--UNUSED WORD
CALLF GATEKEEPER	
POP ES	

RET

;*** TICKET *** TICKET *** TICKET *** TICKET *** TICKET *****/

TICKET:

PUSH ES	
PUSH ES	;TICKET NUMBER DUMMY STORAGE
MOV CX,SP	;POINTER TO TICKET NUMBER
MOV BX,[BX]	;BX <-- PTR TO TICKET NAME
MOV AL,TICKET_IND	
PUSH AX	;N <-- TICKET INDICATER
MOV AL,[BX]	
PUSH AX	;BYT <-- TICKET NAME
PUSH AX	;WORDS <-- UNUSED WORD
PUSH SS	;PTR_SEG <-- TICKET NUMBER SEG
PUSH CX	;PTR_OFFSET <-- TICKET NUMBER POINTER

```

CALLF GATEKEEPER
POP BX                ;RETRIEVE TICKET NUMBER
POP ES

RET

;*** READ *** READ *** READ *** READ *** READ *** READ *****/

READ:

PUSH ES
PUSH ES                ;EVENT COUNT DUMMY STORAGE
MOV CX,SP              ;POINTER TO EVENT COUNT
MOV BX,[BX]            ;BX <-- PTR TO EVENT NAME
MOV AL,READ_IND        ;N <-- READ INDICATER
PUSH AX
MOV AL,[BX]
PUSH AX                ;BYT <-- EVANT NAME
PUSH AX                ;BYT <-- UNUSED WORD
PUSH SS                ;PTR_SEG <-- EVENT COUNT SEGMENT
PUSH CX                ;PTR_OFFSET <--EVENT COUNT POINTER
CALLF GATEKEEPER
POP BX                ;RETRIEVE EVENT COUNT
POP ES

RET

;*** CREATE_PROC *** CREATE_PROC *** CREATE_PROC *****/

CREATE_PROC:

PUSH ES
MOV SI,14[BX]          ;SI <-- PTR TO PROCESS ES
PUSH WORD PTR [SI]     ;STACK PROCESS ES
MOV SI,12[BX]          ;SI <-- PTR TO PROCESS DS
PUSH WORD PTR [SI]     ;STACK PROCESS DS
MOV SI,10[BX]          ;SI <-- PTR TO PROCESS CS
PUSH WORD PTR [SI]     ;STACK PROCESS CS
MOV SI,8[BX]           ;SI <-- PTR TO PROCESS IP
PUSH WORD PTR [SI]     ;STACK PROCESS IP
MOV SI,6[BX]           ;SI <-- PTR TO PROCESS SS
PUSH WORD PTR [SI]     ;STACK PROCESS SS
MOV SI,4[BX]           ;SI <-- PTR TO PROCESS SP
PUSH WORD PTR [SI]     ;STACK PROCESS SP
MOV SI,2[BX]           ;SI <-- PTR TO PROCESS PRIORITY
MOV AH,[SI]            ;GET PROCESS PRIORITY
MOV SI,[BX]            ;SI <-- PTR TO PROCESS ID
MOV AL,[SI]            ;GET PROCESS ID
PUSH AX                ;STACK PROCESS PRIORITY AND ID
MOV CX,SP              ;POINTER TO DATA
MOV AL,CREATE_PROC_IND

```

```

PUSH AX          ;N <-- CREATE PROCESS IND
PUSH AX          ;BYT <-- UNUSED WORD
PUSH AX          ;WORDS <-- UNUSED WORD
PUSH SS          ;PROC_PTR SEGMENT <-- STACK SEG
PUSH CX          ;PROC_PTR OFFSET <-- DATA POINTER
CALLF GATEKEEPER
ADD SP,14        ;REMOVE STACKED DATA
POP ES

```

```
RET
```

```
;*** PREEMPT *** PREEMPT *** PREEMPT *** PREEMPT *****/
```

```
PREEMPT:
```

```

PUSH ES          ;BX <-- PTR TO NAME OF PROCESS
MOV BX,[BX]
MOV AL,PREEMPT_IND
PUSH AX          ;N <-- PREEMPT INDICATER
MOV AL,[BX]
PUSH AX          ;BYTE <-- PREEMPT PROCESS NAME
PUSH AX          ;WORDS <-- UNUSED WORD
PUSH AX          ;PTR_SEG <-- UNUSED WORD
PUSH AX          ;PTR_OFFSET <-- UNUSED WORD
CALLF GATEKEEPER
POP ES

```

```
RET
```

```
;***      DEFINE_CLUSTER ***      DEFINE_CLUSTER ***      **/
```

```
DEFINE_CLUSTER:
```

```

PUSH ES          ;BX <-- PTR TO LOCAL$CLUSTER$ADDR
MOV BX,[BX]
MOV AL,DEFINE_CLUSTER_IND
PUSH AX          ;N <-- DEFINE_CLUSTER_IND
PUSH AX          ;BYT <-- UNUSED WORD
PUSH WORD PTR [BX] ;WORDS <-- LOCAL$CLUSTER$ADDR
PUSH AX          ;PTR_SEG <-- UNUSED WORD
PUSH AX          ;PTR_OFFSET <-- UNUSED WORD
CALLF GATEKEEPER
POP ES
RET

```

```
;*** DISTRIBUTION_MAP *** DISTRIBUTION_MAP *** **/
```

```
DISTRIBUTION_MAP:
```

```
PUSH ES
MOV SI, 4[BX]           ;SI <-- PTR TO GROUP ADDRESS
PUSH WORD PTR [SI]      ;STACK THE GROUP ADDRESS
MOV SI, 2[BX]           ;SI <-- PTR TO ID OF MAP_TYPE
MOV AH, [SI]
MOV SI, [BX]            ;SI <-- PTR TO MAP_TYPE
MOV AL, [SI]            ;AL <-- MAP_TYPE
PUSH AX                 ;STACK ID AND MAP_TYPE
MOV CX, SP              ;POINTER TO DATA
MOV AL, DISTRIBUTION_MAP_IND
PUSH AX                 ;N <-- DISTRIB_MAP_IND
PUSH AX                 ;BYT <-- UNUSED WORD
PUSH AX                 ;WORD <-- UNUSED WORD
PUSH SS                 ;MAP_PTR_SEG <-- SS
PUSH CX                 ;MAP_PTR_OFFSET <-- DATA PTR
CALLF GATEKEEPER
ADD SP, 4
POP ES
RET
```

```
;*** ADD2BIT16 *** ADD2BIT16 *** ADD2BIT16 *** ADD2BIT16 **/
```

```
ADD2BIT16:
```

```
MOV SI,[BX]             ;SI <-- PTR TO BIT(16)#1
MOV BX,2[PX]            ;BX <-- PTR TO BIT(16)#2
MOV BX,[BX]             ;BX <-- BIT(16)#2
ADD BX,[SI]             ;BX <-- BIT(16)#1 + BIT(16)#2
```

```
RET
```

```
END
```

```
;*****
```


APPENDIX E

THE DEMONSTRATION PROGRAM

The demonstration program is a combination of four modules that simulate gathering and processing track information and calculating direction and launcher number for missile launchers.

It was not a goal of this demonstration program to provide a valid algorithm for a real system, but rather to show the synchronization of distributed asynchronous processes with different execution times, working with different length shared data queues, that reside at different addresses at different clusters. Another goal was to demonstrate the systems ability for processes working in shared buffers, or with local copies of shared data items, or a combination of both.

Process `trkdetec` simulates the track detection by producing `x`, `y`, and `z` track data. The track information is put into the 50 slot queue `TRACK`.

Process `trkrprt` simulates the track movement calculation by producing `dx`, `dy`, and `dz` values from the comparison of two consecutive track informations. It consumes `TRACK` data and puts delta information into the 20 slot queue `DELTA`.

Process `mslorder` simulates the missile launcher direction calculation. It consumes `TRACK` data and `DELTA` data and combines these into simulated azimuth and elevation information and assigns a launcher. These data are put into the 50 slot queue `MISSILE_ORDER`.

Process `msltrain` simulates the missile launcher training by consuming `MISSILE_ORDER` data and displaying launcher number, launch number, azimuth, and elevation.

For demonstration purposes each process displays its calculated values and its status (e.g. waiting, going, done,

queue full, etc.)

The program was successfully run in three different constellations:

1.) Process trkdetec and process msltrain multiplexed on one SBC, and process trkrprt and process mslorder multiplexed on a second SBC in the same cluster (cluster 2).

2.) Process trkdetec and process msltrain multiplexed on one SBC in cluster 1, and process trkrprt and process mslorder multiplexed on one SBC in cluster 2.

3.) Process trkdetec on one SBC and process msltrain on another SBC in cluster 1, and process trkrprt on one SBC and process mslorder on another SBC in cluster 2.

The Link86 Input option was used to link cluinit, trkdetec, msltrain, and gatedmod into CLUSERS; c2uinit, trkrprt, mslorder, and gatedmod into C2USERS for constellation 1.) and 2.).

For constellation 3.) following linkage was done:

trkdinit, trkdetec, and gatedmod into TRACKER,
msltinit, msltrain, and gatedmod into MSLREACT,
trkrinit, trkrprt , and gatedmod into REPORTER, and
mslloinit, mslorder, and gatedmod into MSLORD.

This demonstration program, even though done by one person, was built under a simulated lead programmer team policy. The lead programmer provided the system-wide shared data declaration, the system definitions, the cluster specific relation data, pointer assignments, and systems initialization. The lead programmer also decided about the distribution of the different modules over the system.

The application programmers built their modules %including the share.dcl and pointer.ass files.

AD-A164 868

PROCESS SYNCHRONIZATION AND DATA COMMUNICATION BETWEEN
PROCESSES IN REAL TIME LOCAL AREA NETWORKS(U) NAVAL
POSTGRADUATE SCHOOL MONTEREY CA R HAEGER DEC 85

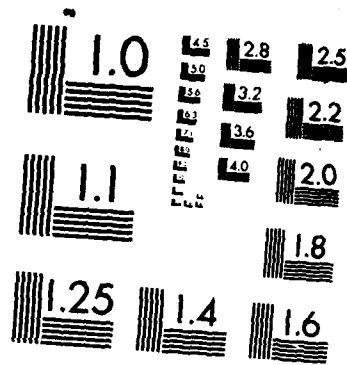
2/2

UNCLASSIFIED

F/G 17/2

ML

ENC
10MB
1.5
END



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

```

*****
*****
***          C1USERS.INP file          ***
***-----***
*** This file is used to link user initialization, ***
*** user process msltrain, user process trkdetec, and ***
*** gatemodule into C1USERS.CMD, the executable file ***
*** multiplexing the two user processes on one SBC. ***
*****

```

```

ciusers =
ciuinit [code[ab[439]].data[ab[800].m[0],ad[82]].map[all]].
trkdetec,
msltrain,
gatemod

```

```

*****
*****
***          C1UINIT.PLI file          ***
***-----***
*** This is the initialization procedure for the ***
*** multiplexed user constellation. ***
*****

```

```

c1_users_init: procedure options (main);

    %include 'sysdef.pli';

    /* begin */

        /* trkdetec */
        call create_proc ('05'b4, 'fc'b4,
                        '09eb'b4, '0800'b4, '0029'b4,
                        '0439'b4, '0800'b4, '0800'b4);

        /* msltrain */
        call create_proc ('06'b4, 'fc'b4,
                        '0a0b'b4, '0800'b4, '0205'b4,
                        '0439'b4, '0800'b4, '0800'b4);
        call await ('fe'b4, '01'b4);
    end c1_users_init;

```

```

*****

```

```

*****
*****
***          TRACKER.INP file          ***
***-----
*** This file is used to link trkdetec initialization, ***
*** user process trkdetec, and gatemodule into.      ***
*** TRACKER.CMD, the executable file for user process ***
*** trkdetec in the non-multiplexed constellation.    ***
*****

```

```

tracker =
trkdnit [code [ab[439]],data[ab[800],m[0],ad[82]],map[all]],
trkdetec,
gatemod

```

```

*****
*****
***          TRKDINIT.PLI file          ***
***-----
*** This is the initialization procedure for user      ***
*** process trkdetec in the non-multiplexed          ***
*** constellation.                                    ***
*****

```

```

trkdnit:  procedure options (main);

%include 'sysdef.pli';

/* begin */
    call create_proc ('01'b4, 'fc'b4,
                     '085d'b4, '0800'b4, '0023'b4,
                     '0439'b4, '0800'b4, '0800'b4);
    call await ('fe'b4, '01'b4);

end trkdnit;

```

```

*****
*****
***          MSLREACT.INP file          ***
***-----
*** This file is used to link msltrain initialization, ***
*** user process msltrain, and gatemodule into        ***
*** MSLREACT.CMD, the executable file for user process ***
*** msltrain in the non-multiplexed constellation.    ***
*****

```

```

mslreact =
msltnit [code[ab[439]],data[ab[800],m[0],ad[82]],map[all]],
msltrain,
gatemod

```

```

*****
*****
***      MSLTINIT.PLI file      ***
***-----***
*** This is the initialization procedure for user      ***
*** process msltrain in the non-multiplexed            ***
*** constellation.                                     ***
*****
*****

```

```
msltinit:  procedure options (main);
```

```
  %include 'sysdef.pli';
```

```
  /* begin */
```

```

      call create_proc ('02'b4, 'fc'b4,
                        '0825'b4, '0800'b4, '0023'b4,
                        '0439'b4, '0800'b4, '0800'b4);
      call await ('fe'b4, '01'b4);
end msltinit;

```

```

*****
*****
***      TRKDETEC.PLI file      R. Haeger, Dec 1985      ***
***-----***
*** This is the PL/I-96 code for user process trkdetec. ***
*** It simulates track detection by incrementing track ***
*** position values every iteration. It produces shared ***
*** data TRACK.                                           ***
*****
*****

```

```
trkdetect:  procedure ;
```

```
  %replace
```

```

      infinity      by 32767,
      one           by '0001'b4,
      tolen         by 50;

```

```

  %include 'sysdef.pli';
  %include 'share.dcl';

```

```
  /* used shared data:
```

```

      1 track(0:49) based(tr_ptr),
      2 x fixed bin (15),
      2 y fixed bin (15),
      2 z fixed bin (15),      */

```

```

DECLARE

    i fixed bin (15),
    (k,tq_ub,tq_lb) bit (16),
    1 local_track,
    2 x fixed bin (15),
    2 y fixed bin (15),
    2 z fixed bin (15);

/* main */
%include 'pointer.ass';

do i = 0 to infinity;

/* simulation of track input data */
    local_track.x=i+1;
    local_track.y=i+10;
    local_track.z=i+1;

/* put track in shared memory */
    track(mod(i,talen)) = local_track;
    call advance (TRACK_IN);
    tq_ub = read(TRACK_IN);

/* display track values */
    put skip(2) edit ('Track ',binary(tq_ub),
                     ' x: ',local_track.x,
                     ' y: ',local_track.y,
                     ' z: ',local_track.z,
                     ' put in queue slot ',mod(i,talen))
                     (4(a,f(5)));
    tq_lb = read (TRACK_OUT);

/* report status */
    put skip(2) edit('Last consumed track ',binary(tq_lb),
                     ' in slot: ',mod(binary(tq_lb)-1,talen))
                     (2(a,f(5)));

    /* check if slot available for next iteration */
    if ((binary(tq_ub)-binary(tq_lb)) >= talen ) then
do;
        k = add2bit16(tq_lb,one);
        /* report status */
        put skip(2) edit ('Waiting for slot: ',
                          mod(binary(k)-1,talen),
                          ' to be consumed ') (a,f(3),a);
        call await (TRACK_OUT, k);
    end;
end; /* do FOREVER */

end trkdetect;

```



```

*****
*****
***      MSLTRAIN.PLI file      R. Haeger, Dec 1985      ***
***-----***
*** This is the PL/I-86 code for user process msltrain. ***
*** It simulates missile launcher training by          ***
*** displaying launcher assignment and direction values. ***
*** It consumes shared data MISSILE_ORDER.             ***
*****
*****

```

```
msltrain:  procedure ;
```

```
  %replace
```

```

    infinity      by 32767,
    one           by '0001'b4,
    moqlen        by 50;

```

```
  %include 'sysdef.pli';
```

```
  %include 'share.dcl';
```

```
  /* used shared data:
```

```

    1 missile_order(0:49) based(mo_ptr),
    2 launcher fixed bin (7),
    2 azimuth float binary,
    2 elevation float binary,          */

```

```
  DECLARE
```

```

    i fixed bin (15),
    k bit (16) static init ('0000'b4);
  /* end DECLARATIONS */

```

```
  /* main */
```

```
  %include 'pointer.ass';
```

```
  do i = 0 to infinity;
```

```
    k = add2bit16(k, one);
```

```
    /* report status */
```

```
    put skip list('msltrain waiting');
```

```
    call await (MISSILE_ORDER_IN, k);
```

```
  /* consume and display missile order values */
```

```

    put skip(2) edit('launcher: ',
                     missile_order(mod(i,moqlen)).launcher,
                     ' launch: ',binary(k),
                     ' azimuth: ',
                     missile_order(mod(i,moqlen)).azimuth,
                     ' elevation: ',
                     missile_order(mod(i,moqlen)).elevation)
              (2(a,f(5)),2(a,e(10,2))));

```

```

        call advance (MISSILE_ORDER_OUT);

    end; /* do i */

end msltrain;

*****
*****
***          C2USERS.INP file          ***
***-----
*** This file is used to link user initialization, ***
*** user process mslorder, user process trkrprt, and ***
*** gatmodule into C2USERS.CMD, the executable file ***
*** multiplexing the two user processes on one SBC. ***
*****

c2users =
c2uinit [code[ab[439]].data[ab[800].m[0].ad[82]].map[all]].
mslorder,
trkrprt,
gatmod

*****
*****
***          C2UINIT.PLI file          ***
***-----
*** This is the initialization procedure for the ***
*** multiplexed user constellation. ***
*****

c2_users_init: procedure options (main);

    %include 'sysdef.pli';

    /* begin */

        /* missile order */
        call create_proc ('03'b4, 'fc'b4,
                           '0a29'b4, '0800'b4, '0029'b4,
                           '0439'b4, '0800'b4, '0800'b4);

        /* track report */
        call create_proc ('04'b4, 'fc'b4,
                           '0b47'b4, '0800'b4, '0303'b4,
                           '0439'b4, '0800'b4, '0800'b4);

        call await ('fe'b4, '01'b4);
    end c2_users_init;

*****

```

```

*****
*****
***      REPORTER.INP file      ***
***-----***
*** This file is used to link trkrprt initialization, ***
*** user process trkrprt, and gatemodule into ***
*** REPORTER.CMD, the executable file for user process ***
*** trkrprt in the non-multiplexed constellation ***
*****

```

```

reporter=
trkrinit [code[ab[439]],data[ab[800],m[0],ad[82]],map[all]],
trkrprt,
gatemod

```

```

*****
*****
***      TPKRINIT.PLI file      ***
***-----***
*** This is the initialization procedure for user ***
*** process trkrprt in the non-multiplexed constellation.***
*****

```

```

trkrinit: procedure options (main);

```

```

    %include 'sysdef.pli';
    /* begin */

```

```

        call create_proc ('04'b4, 'fc'b4,
                           '087f'b4, '0800'b4, '0023'b4,
                           '0439'b4, '0800'b4, '0800'b4);

```

```

        call await ('fe'b4, '01'b4);

```

```

end trkrinit;

```

```

*****
*****
***      MSLORD.INP file      ***
***-----***
*** This file is used to link mslorder initialization, ***
*** user process mslorder, and gatemodule into ***
*** MSLORD.CMD, the executable file for user process ***
*** mslorder in the non-multiplexed constellation. ***
*****

```

```

mslord =
msloinit [code[ab[439]],data[ab[800],m[0],ad[82]],map[all]],
mslorder,
gatemod

```

```

*****

```

```

*****
*****
***      MSLOINIT.PLI file      ***
***-----***
*** This file is the initialization procedure for user ***
*** process mslorder in the non-multiplexed constellation ***
*****

```

```
msloinit: procedure options (main);
```

```
  %include 'sysdef.pli';
```

```
  /* begin */
```

```
    call create_proc ('03'b4, 'fc'b4,
                      '09b3'b4, '0000'b4, '0023'b4,
                      '0439'b4, '0000'b4, '0800'b4);
```

```
    call await ('fe'b4, '01'b4);
```

```
end msloinit;
```

```

*****
*****
***      TRKRPRT.PLI file      R. Haeger, Dec 1985      ***
***-----***
*** This is the PL/I-86 code for user process trkrprt. ***
*** It simulates computation of delta values for tracks ***
*** by comparing two consecutive positions of a track. ***
*** It consumes shared data TRACK and produces shared ***
*** data DELTA. ***
*****

```

```
trkrprt: procedure ;
```

```
  %replace
```

```

        infinity      by 32767,
        one            by '0001'b4,
        dolen          by 20,
        tolen          by 50;
```

```
  %include 'sysdef.pli';
  %include 'share.dcl';
```

```
  /* used shared data:
```

```

    1 track(0:49) based(tr_ptr),
    2 x fixed bin (15),
    2 y fixed bin (15),
    2 z fixed bin (15),
```

```

1 delta(0:19) based(de_ptr),
2 dx fixed bin (7),
2 dy fixed bin (7),
2 dz fixed bin (7),    */

DECLARE

1 fixed bin (15),
(km,k) bit (16) static init ('0000'b4),

(dq_ub,dq_lb) bit (16),

1 local_track,
2 x fixed bin (15),
2 y fixed bin (15),
    'waiting for slot ',
    mod(binary(k)-1,dqlen),
    ' to be con          2 dy fixed bin (7),
2 dz fixed bin (7);

/* end DECLARATIONS */

/* main */
#include 'pointer.ass';

do i = 0 to infinity;
    /* report status */
    put skip(2) list ('proc trkrprt, iteration: ',
        i, ' waiting ');

    km = add2bit16(km, one);
    call await (TRACK_IN, km);
    /* report status */
    put list(' going ');

/* read shared data item and compute delta values */

local_delta.dx=(track(mod(i,tolen)).x)-(local_track.x);
local_delta.dy=(track(mod(i,tolen)).y)-(local_track.y);
local_delta.dz=(track(mod(i,tolen)).z)-(local_track.z);

/* display computed delta values */
put skip list(' dx:',local_delta.dx,
    ' dy:',local_delta.dy,
    ' dz:',local_delta.dz);

/* save track data for next iteration */
local_track=track(mod(i,tolen));

/* put delta in shared memory */
delta(mod(i,dqlen))=local_delta;

```

```

call advance(DELTA_IN);

    /* report status */
    put list(' done ');

dq_ub=read(DELTA_IN);
dq_lb=read(DELTA_OUT);

/* check if slot available for next iteration */
if ((binary(dq_ub)-binary(dq_lb)) >= dqlen) then
do;
    k=add2bit16(dq_lb,one);
    /* if queue is full, report status */
    put skip(2) edit('Delta queue full, ',
                    'waiting for slot ',
                    mod(binary(k)-1,dqlen),
                    ' to be consumed')
                    (a,a,f(3),a);

    call await(DELTA_OUT,k);
end;
end; /* do 1 */

end trkrprt;

```

```

*****
*****
***      MSORDER.PLI file      R. Haezer, Dec 1985      ***
***-----***
*** This is the PL/I-36 code for user process mslorder. ***
*** It simulates computation of missile launcher        ***
*** direction and launcher assignment.                   ***
*** It consumes shared data TRACK and DELTA, and         ***
*** produces shared data MISSILE_ORDER.                  ***
*****

```

```

mslorder:  procedure ;

```

```

%replace

```

infinity	by 32767,
one	by '0001'b4,
talen	by 50,
dolen	by 20,
modlen	by 50;

```

%include 'sysdef.pli';
%include 'share.dcl';

```

```

/* used shared data:
  1 track(0:49) based(tr_ptr),
    2 x fixed bin (15),
    2 y fixed bin (15),
    2 z fixed bin (15),

  1 delta(0:19) based(de_ptr),
    2 dx fixed bin (7),
    2 dy fixed bin (7),
    2 dz fixed bin (7),

  1 missile_order(0:49) based(mo_ptr),
    2 launcher fixed bin (7),
    2 azimuth float binary,
    2 elevation float binary, */

```

DECLARE

```

  i fixed bin (15),
  km bit (16) static init ('0000'b4),
  kd bit (16) static init ('0000'b4),
  (moq_ub,moq_lb) bit (16),

  1 local_track,
    2 x fixed bin (15) ,
    2 y fixed bin (15) ,
    2 z fixed bin (15) ,

  1 local_delta,
    2 dx fixed bin (7),
    2 dy fixed bin (7),
    2 dz fixed bin (7),

  1 local_order,
    2 launcher fixed bin (7),
    2 azimuth float binary,
    2 elevation float binary;

```

/* end DECLARATIONS */

/* main */

```

%include 'pointer.ass';
do i = 0 to infinity;

```

```

  /* report status */
  put skip(2) list('proc mslorder, iteration: ',
    i, ' m_waiting ');

```

```

  kd=add2bit16(kd,one);
  call await(DELTA_IN,kd);

```

```

    /* report status */
    put list(' m_going ');
    /* copy track values */
    local_track=track(mod(i,talen));
    call advance(TRACK_OUT);
    /* copy delta values */
    local_delta=delta(mod(i,dalen));
    call advance(DELTA_OUT);
    /* display track values */
    put skip list(' x:',local_track.x, ' y:',local_track.y,
                  ' z:',local_track.z);
    /* assign launcher */
    local_order.launcher=mod(i,4)+1;
    /* simulate direction computation */
    local_order.azimuth=
        atand(float(local_track.y + local_delta.dy)/
              float(local_track.x + local_delta.dx));
    local_order.elevation=
        atand(float(local_track.z + local_delta.dz)/
              float(local_track.x + local_delta.dx));
    /* put missile order in shared memory */
    missile_order(mod(i,modlen))=local_order;
    /* display missile order values */
    put skip list(' l:',local_order.launcher,
                  ' a:',local_order.azimuth,
                  ' e:',local_order.elevation);
    call advance(MISSILE_ORDER_IN);

    /* report status */
    put list(' m_done ');
    mod_ub = read(MISSILE_ORDER_IN);
    mod_lb = read(MISSILE_ORDER_OUT);

    /* check if slot available for next iteration */
    if ((binary(mod_ub)-binary(mod_lb))>=modlen ) then
    do;
        km = add2bit16(mod_lb,one);
        /* report if queue is full */
        put skip(2) edit('Missile_order queue full, ',
                        'waiting for slot ',
                        mod(binary(km)-1,modlen),
                        ' to be consumed')
                        (a,a,f(3),a);
        call await (MISSILE_ORDER_OUT, km);
    end;

end; /* do i */

end mslorder;

*****

```


APPENDIX F
SYSTEM INITIALIZATION

To switch on and initialize the system, follow these steps:

(1)

Switch on and set up the hardware components of both clusters in accordance with the respective power on procedures described in the AEGIS lab up to the point where the system disks are in their drives and the reset button of the MULTIBUS frame was pushed.

(2)

Insert MCORTEX disks for cluster 1 and cluster 2 in their respective drives.

(3)

For cluster 1:

At terminal 1 type in: capital 'U'.

After prompt type in: 'gffd4:4' followed by RETURN.

After prompt choose console '1' and login disk 'B'.

After prompt B> change to disk A.

After prompt A> type in: 'ldcpm'.

After prompt A> type in: 'ldboot'.

After prompt A> change to disk B.

After prompt B> terminal 1 is ready for MCORTEX.

At terminal 2 type in: capital 'U'.

After prompt type in: 'ge000:400'.

After prompt choose console '2' and login disk 'C'.

After prompt C> change to disk B.

After prompt B> terminal 2 is ready for MCORTEX.

At terminal 3 type in: capital 'U'.

After prompt type in: 'ge000:400'.
After prompt choose console '3' and login disk 'D'.
After prompt D> change to disk B.
After prompt B> terminal 3 is ready for MCORTEX.

For cluster 2:

At terminal 1 type in: capital 'U'.
After prompt type in: 'gffd4:0'.
After prompt do the same as in cluster 1 after this step.

Make sure cluster 1 is connected to the RTC* Ethernet.
System is ready for MCORTEX.

(4)

At cluster 1:

At terminal 1 type in: 'MCORTEX' followed by RETURN.
System will ask for global memory to be loaded, type in:
'Y' and hit RETURN.
System will ask for filename.

At terminal 2 type in: 'MCORTEX' followed by RETURN.
System will ask for global memory to be loaded, hit
RETURN.
System will ask for filename.

At terminal 3 do the same as at terminal 2.

Cluster 1 is ready for initialization.

At terminal 1 type in: 'C1PROC.CMD' and hit RETURN.
The driver will prompt with length of longest queue, and
signal that cluster 1 is initialized.
Cluster 1 is ready for the application processes.

Initialize cluster 2 the same way as cluster 1, but type
in: 'C2PROC.CMD' instead of 'C1PROC.CMD'.
After driver prompts, the system is ready for
application processes.

Following, the initialization of the demonstration program's non-multiplexed constellation is described.

(5)

At cluster 2:

At terminal 2 type in: 'MSLORD.CMD'.

User process MSLORDER will start executing.

At terminal 3 type in: 'REPORTER.CMD'.

User process TRKRPRT will start executing.

At cluster 1:

At terminal 3 type in: 'MSLREACT.CMD'.

User process MSLTRAIN will start executing.

At terminal 2 type in: 'TRACKER.CMD'.

User process TRKDETEC will start executing.

The total system will run.

(6)

Every terminal displays respective data on its screen.

Terminals 1 in both clusters show message exchange activity 'transmitting' or 'receiving'.

(7)

To stop any process, hit 'Control S' at the respective terminal. To restart the process hit 'Control S' again.

LIST OF REFERENCES

1. Brewer, D.J., A Real-Time Executive for Multiple-Computer Clusters, M.S. Thesis, Naval Postgraduate School, Monterey, California, December 1984
2. Swan, R.J., and others, CM* - A Modular Multi-Microprocessor, Proceedings of the National Computer Conference, 1977.
3. Reed, D.P. and Kanodia, R.J., "Synchronization with Eventcounts and Sequencers," Communication of the ACM, Volume 22, p. 115-123, February 1979.
4. Wasson, W.J., Detailed Design of the Kernel of a Real-Time Multiprocessor Operating System, M.S. Thesis, Naval Postgraduate School, Monterey, California, June 1980.
5. Rapantzikos, D.K., Detailed Design of the Kernel of a Real-Time Multiprocessor Operating System, M.S. Thesis, Naval Postgraduate School, Monterey, California, March 1981.
6. Cox, E.R., A Real-Time, Distributed Operating System, M.S. Thesis, Naval Postgraduate School, Monterey, California, December 1981.
7. Klinefelter, S.G., Implementation of a Real-Time, Distributed Operating System for a Multi-Computer System, M.S. Thesis, Naval Postgraduate School, Monterey, California, June 1982.
8. Rowe, W.R., Adaption of MCORTEX to the AEGIS Simulation Environment, M.S. Thesis, Naval Postgraduate School, Monterey, California, June 1984.
9. Xerox Corporation, The Ethernet - A Local Area Network: Data Link and Physical Layer Specification, Version 1.0, September 1980.
10. InterLAN Corporation, NI3010 MULTIBUS Ethernet Communication Controller User Manual, 1982.
11. Digital Research, PL/I Language Reference Manual, 1982.

INITIAL DISTRIBUTION LIST

	No.	Copies
1. Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2	
2. Library, Code 0142 Naval Postgraduate School Monterey California 93943	2	
3. Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943	1	
4. CDR Gary S. Baker, Code 52Bj Department of Computer Science Naval Postgraduate School Monterey, California 93943	1	
5. Dr. Uno R. Kodres, Code 52Kr Department of Computer Science Naval Postgraduate School Monterey, California 93943	3	
6. LCDR Reinhard Haeger Ostendstrasse 20 4750 Unna-Massen West-Germany	3	
7. Daniel Green, Code 20F Naval Surface Weapons Center Dahlgren, Virginia 22449	1	
8. CAPT J. Donegan, USN PMS 400B5 Naval Sea Systems Command Washington, D.C. 20362	1	
9. PCA AEGIS Data Repository RCA Corporation Government Systems Division Mail Stop 127-327 Moorestown, New Jersey 08057	1	
10. Library (Code E33-05) Naval Surface Warfare Center Dahlgren, Virginia 22449	1	
11. Dr. M. J. Gralia Applied Physics Laboratory John Hopkins Road Laurel, Maryland 20707	1	
12. Dana Small Code 8242, NOSC San Diego, California 92152	1	
13. Dokumentationszentrum der Bundeswehr Friedrich-Ebert-Allee 34 5300 Bonn West-Germany	1	

DTIC

FILMED

4-86

END